

所见的是暂时的，所不见的是永远的

计算机的心智 操作系统 之哲学原理

Computer's Mind
Philosophical Principles of Operating Systems

邹恒明 著



机械工业出版社
China Machine Press

所见的是暂时的，所不见的是永远的

计算机的心智 操作系统 之哲学原理

Computer's Mind

Philosophical Principles of Operating Systems

邹恒明 著



机械工业出版社
China Machine Press

本书集中精力对操作系统的核心内容进行分析,包括操作系统发展的历史背景、进程与线程、内存管理、文件系统、输入与输出、多核环境下的进程调度和操作系统设计。本书用大量生活实例,生动解释了操作系统中的主要难点和模糊点:锁的实现、同步机制的发展轴线、纯粹分段到段页式的演变、多核环境下的进程同步与调度和操作系统设计等内容,而放弃了对操作系统核心以外内容,如安全、多媒体系统、虚拟机技术、光盘技术等论述。本书重点突出、逻辑清晰、内容连贯,便于学生顺利掌握操作系统的核心内容。

本书层次丰富、涵盖操作系统的所有核心内容,适合作为国内高校计算机及相关专业本科生操作系统课程的教材,也是了解计算机操作系统原理不可多得的参考书。

版权所有,侵权必究。

本书法律顾问 北京市展达律师事务所

图书在版编目(CIP)数据

计算机的心智:操作系统之哲学原理/邹恒明著. —北京:机械工业出版社, 2009. 4

ISBN 978-7-111-26642-6

I. 计… II. 邹… III. 操作系统 IV. TP316

中国版本图书馆 CIP 数据核字(2009)第 041558 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑:金 纯

北京京北印刷有限公司印刷

2009 年 4 月第 1 版第 1 次印刷

186mm × 240mm · 20.25 印张

标准书号:ISBN 978-7-111-26642-6

定价:38.00 元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换

本社购书热线:(010) 68326294

前言 PREFACE



In Pursuit of Absolute Simplicity 求于至简，归于永恒

谨以此书献给夫人蕾蕾，女儿雨洁、雨蓉、雨恒和雨宜。

当你在电脑上玩游戏的时候，当你在电脑上与朋友聊天的时候，当你编写完一个程序需要加载运行的时候，你有没有一种像在观看魔术的感觉？编写好的程序能够编译运行，计算出结果，并显示或打印出来。你有没有觉得它很神秘？

如果想揭开这层神秘的面纱，你就得学习操作系统。

因为操作系统是掌控计算机运行的系统，在学习它的过程中，读者能够了解到程序在计算机上运行的全景，或者说我们所认为的全景。之所以这么说，是因为精确了解程序在计算机上运行的全景是极其困难的（有人认为这根本就是不可能的）。当然，这里的程序指的是有一定规模的程序，而不是那种只有几行代码的小程序。从某种程度上来说，没有人敢肯定自己清楚计算机在任何一个时刻所处的状态。例如，在多流水线计算机上，如果发生中断或异常，我们根本就得不到一个精准的状态。唯一能做的就是推倒重来。



图1 风靡世界的游戏“第二生命”

计算机的心智

人有心智吗？我想所有人都会回答：有！人的心智就是人的灵气。这是每一个人的生命之气。就是这个灵气赋予了人丰富的思维、感受和行动能力。

那么计算机有心智吗？这不是一个诡秘或者搞笑的问题。

人们通常认为能够运动的生命都是有灵气的，既然计算机能够完成一些人脑才能够完成的理性任务，它当然也有心智！而这个心智就是操作系统。因为操作系统赋予了计算机以活力。虽然读者有可能尚不明白操作系统是怎么一回事，但也许知道没有操作系统，现代计算机是运转不起来的（这里需排除远古时代的古老计算机）。操作系统作为计算机赖以运转的控制中心，称其为计算机的心智可谓恰如其分。



图2 计算机的心智就是操作系统

操作系统的奥秘

记得小时候常常念的一首诗是这样的：

从小时候就开始数了。
数到懂事、数到成熟，
还没有数清。
天上的星星为什么数不清呢？
像记忆和幻想，
永远背负着固执的迷……

对于许多大学是学计算机及相关专业的同学来说，操作系统就像是天上的星星（如图3所示），隐藏着一个固执的迷，永远学不清楚。不过，操作系统真的难以学清楚吗？



图3 理解操作系统有点类似于数清楚天上的星星

不是的。学不清楚是因为没有看到其背后的奥秘。这个奥秘不是所有人都知道的。即使是研究操作系统的人也不一定意识到它，初学计算机者自然就更加不会注意了。

那么这个奥秘是什么呢？

天上的星星数不清是因为我们试图做的事情是数星星。如果我们换个角度，不去数星星，而是寻找到星星的设计师，让他告诉我们星星的数量，不就数清楚了吗？

这也正是学习操作系统的奥秘。要理解操作系统，就要寻找到操作系统的设计师们，让他们告诉我们操作系统所蕴含的所有秘密。当然，这里的寻找设计师并不是真的找来他们，因为找到所有的设计师是不可能的。这里的设计师指的是一种抽象，一种所有设计师所共有的人生哲学，因为设计师们在设计操作系统时会不自觉地将自己的思维或人生追求构造在操作系统里，从而赋予了操作系统以心智，而操作系统也就在这种心智的指挥下亘古运行着。

操作系统之哲学原理

正如前面所述，让设计师告诉我们操作系统秘密是理解操作系统的最好办法。他们所用的载体就是其所遵循的生活哲学，这些生活哲学就是操作系统所遵循的哲学原理。

本书就是试图从这些哲学原理（也就是人类生活哲学）的视角来阐述操作系统，从而揭

开操作系统的神秘面纱，令其不再晦涩难懂。

例如，CPU 管理（进程与线程）、内存管理（虚拟存储）、外存管理（文件系统）、I/O 管理（输入与输出）等操作系统的核心机制不外乎是资源的管理，它们都遵循着一切人类资源管理的基本原则，即如何有效地发掘资源、监控资源、分配资源和回收资源。

除了提供管理的功能外，操作系统还需要保证自己的正常运转，即它必须尽力使自身不发生失效或崩溃，因为这是提供其他一切功能的基础。这与人类把确保自身健康生活作为开发利用资源的前提是一个道理。

如果我们把握了资源的根本属性，即资源管理必然涉及共享和竞争的管理，理解了操作系统必须首先保障自己的正常运转，就会理解操作系统的一切行为。前者指引着操作系统功能的设计与进化，后者则推动着操作系统可靠性地演变。

资源管理也好，保证自身的正确性也好，它们都有着根本的线索。这条根本线索就是人类在长期的生活实践中摸索出来的管理社会和保障自身安全的各种办法。这些办法是随着人类哲学思维的变化而改进的。因此，只要明白了人类的哲学思维，就能明白操作系统所遵循的哲学原理，进而明白整个操作系统的设计与构造。

除了使操作系统易于理解外，从哲学的层面阐述操作系统的原理还有如下好处：

- 操作系统可以变化，但支持其存在的哲学原理是不变的。这样，本书的内容可以在操作系统不断演变的环境下保持有效，而不会像其他书的内容，随着时间的推移而过时。
- 对于很多人来说，操作系统所采取的机制、策略和手段看上去十分枯燥，如果从哲学原理上给它们赋予人性的特点，这些机制、策略和手段便不再枯燥。

通过将人生哲学与操作系统联系起来，从操作系统哲学原理的层次阐述操作系统的核心技术，就能够理解掌握操作系统的精髓。

本书内容安排

为清楚地阐述操作系统的哲学原理，也为了使内容显得紧凑，逻辑上一气呵成，本书只选择了操作系统的核心内容进行分析，放弃了对操作系统核心以外内容，如安全、多媒体系统、虚拟机技术、光盘技术等论述。本书集中精力对操作系统发展的历史背景、进程与线程、内存管理、文件系统、输入与输出、多核环境下的进程调度和操作系统设计进行了哲学原理层面的分析与论述。对内容的这种安排有如下好处：

- 可使本书重点突出、逻辑清晰、内容连贯，便于学生顺利掌握操作系统的核心与关键。
- 操作系统的核心内容经过长久的研究与实践，已经变得较为稳定并且形成了公认的标准，讲解起来没有歧义。
- 操作系统的非核心部分由于研究的时间短，工业界参与的程度较低，并无公认的标准，论述起来要么不全面，要么显得凌乱，使刚刚接触操作系统的读者感到迷惑。
- 只要掌握了核心内容的原理，读者便能通过自学掌握操作系统核心以外的知识。

本书覆盖全国硕士研究生入学统一考试计算机学科专业基础综合考试大纲中操作系统全部内容。

本书一共分为7篇22章。7篇分别是基础原理篇、进程原理篇、内存原理篇、文件原理篇、I/O原理篇、多核原理篇和操作系统设计原理篇。本书的内容结构如图4所示。

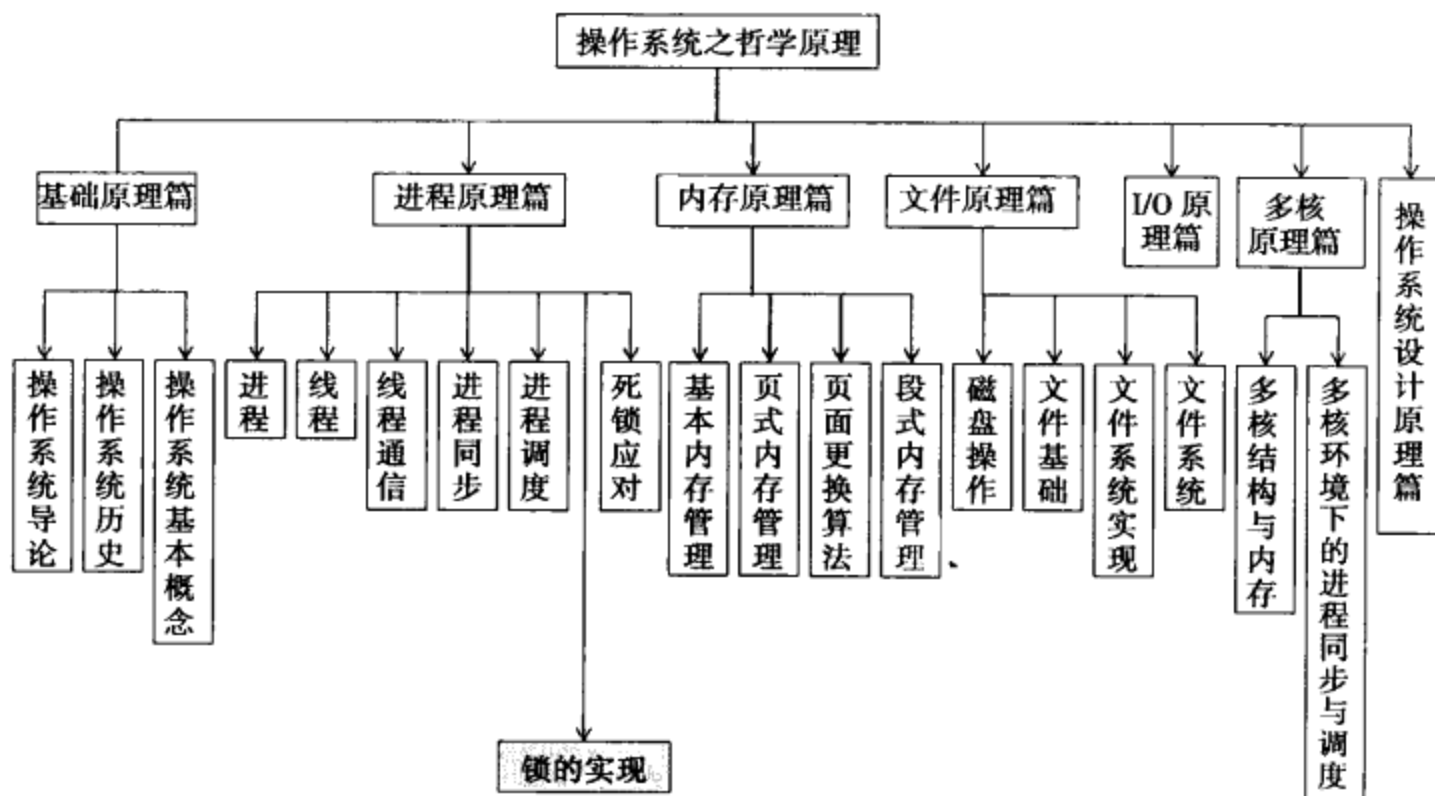


图4 本书内容结构

基础原理篇

该篇包含第1章至第3章的内容。第1章的内容包括智者的挑战、人造学科、程序是如何运行的、什么是操作系统、魔术与管理、用户程序与操作系统、操作系统范畴和为什么学习操作系统。第2章探讨操作系统演变的主要过程：从单一控制终端单一操作员，到批处理、多道批处理、分时操作系统、实时操作系统、现代操作系统；对商业操作系统演变的过程进行分析，然后探讨操作系统分类和操作系统的未来发展趋势。第3章简要回顾计算机硬件基本知识，探讨什么是“抽象”，讲解用户态与内核态，阐述操作系统结构、系统调用、操作系统的壳等知识。

进程原理篇

该篇对操作系统最为核心的概念“进程”进行讲解，包括第4章至第10章的内容。第4章阐述的内容包括进程出现的逻辑必然性、多道编程的效率、进程的创建和消亡、进程的状态及其转换、进程与地址空间、进程管理和进程模型的缺陷。第5章讲解的内容包括线程、线程管理、线程的用户态、内核态和混合态实现、现代操作系统的线程实现模型、多线程之间的关系、线程主要考虑的问题。第6章的内容包括为什么要通信、管道、记名管道、套接字、信号、信号量、共享内存、消息队列等。第7章的内容包括为什么同步、同步的目的、锁原语的进化、睡觉与叫醒原语、信号量、管程、消息传递和栅栏。第8章讲解的内容包括调度的目标、先来先服务、时间片轮转、短任务优先、优先级调度、混合调度、实时调度等算法，并对优先级倒挂和线程的不确定性进行讨论。第9章讲述如何使用中断启用和禁止、测试与设置来实现锁原语。第10

章对死锁的产生、发展、防止与避免进行讲解，并讨论死锁、活锁和饥饿的关系。

内存原理篇

该篇对操作系统的另外一个重要构成部分“内存管理”进行阐述，包括第11章至第14章的内容。第11章讲述内存管理的环境、内存管理的目标、虚拟内存、操作系统在内存中的位置、单道编程的内存管理、固定加载地址、多道编程的内存管理、固定分区、非固定分区、交换、地址翻译、闲置空间管理等内容。第12章的内容包括基址极限的问题、分页管理、页表、页面翻译过程、分页管理系统的优缺点、多级页表、地址翻译速度、锁住页面、内存抖动和页面尺寸设计。第13章对页面更换算法的来龙去脉、欲达到的目的、各种具体的页面更换算法进行细致讲解。第14章的内容包括分段管理系统、分段的优缺点、段号与寻址位数，并对否定之否定在内存管理模式发展过程中的作用进行讨论。

文件原理篇

该篇对操作系统的第三个核心构件“文件系统”进行讲解，包括第15章至第18章的内容。第15章讲述的内容包括磁盘结构、磁盘访问速度、磁盘的操作系统界面、磁盘访问过程和磁盘调度。第16章讲述为什么需要文件系统、什么是文件系统、文件系统的目的、文件的基本知识、文件的存储结构、文件类型、文件访问、文件属性、文件操作、文件夹、相对与绝对路径、共享与链接、内存映射的文件等内容。第17章的内容包括文件系统分布、文件的实现、文件夹的实现、共享文件的实现、磁盘空间的管理等。第18章的内容包括文件安全性能（文件访问控制、访问控制表、能力表）、文件可靠性能（持久性、一致性、日志、交易、随影、一致性检查）和文件系统的效率性能（提前读取、减少磁臂移动距离、日志结构的文件系统LFS）。

I/O 原理篇

该篇对计算机与外界进行沟通的机制“输入与输出”进行讲解。本篇仅有一章（第19章），讨论的内容包括输入输出的重要性和目的、I/O硬件的哲学原理、物理I/O模式（专有通道I/O、内存映射的I/O、复合I/O、DMA）、输入输出软件之哲学原理、软件I/O模式（可编程I/O、中断驱动I/O、DMA）、I/O软件分层和设备驱动程序等。

多核原理篇

该篇对新出现的多核技术进行讲解。重点讨论多核环境给操作系统带来的影响。全篇分为多核结构和多核操作系统两章。第20章讲解的内容包括多核处理器结构（超线程结构、多核结构、多核超线程结构）、多核内存结构（UMA、NUMA、COMA、NORMA）、对称多核处理器计算机的启动过程、多处理器之间的通信和SMP缓存一致性等。第21章的内容包括多核进程同步、多核环境下的软件同步原语、旋锁及其实现、队列旋锁、多核环境下的进程调度、多核环境下的能耗管理和多核系统性能。

操作系统设计原理篇

该篇从高屋建瓴的角度对操作系统设计的十条哲学原理进行阐述。显然，操作系统的设计原则有很多，本篇选取的只是这诸多原则里面非常重要的十条。第22章将从操作系统和人类社会两个层面对这十条原理进行论述与比较，以使读者更加清楚地明白操作系统就是人类社会在计算机里面的反映。操作系统的其他设计原则读者可自行发现。

本书的特点

相对于国内外其它操作系统教材，本书的独特性体现在四个方面：逻辑导向，通过逻辑推理将核心原理演绎出来；联系生活，用人所熟知的生活实例来揭示奥秘；抽象提升，从哲学高度进行阐述以将各种原理串成有机整体；知识整合，引入相关编译和计算机组成的知识来加深读者对细腻之处的把握。这些特点赋予了本书风格上更加清新、内容上更加丰富、逻辑上更加严谨、叙述上更加幽默、解说上更加深刻、和层次上更加优美的引人入胜的效果。

读者在阅读学习完本书后，将达到如下目标：

- 了解操作系统在计算机软硬件整个体系中的中心主导作用。
- 掌握操作系统的基本概念、原理、技术和实现机制。
- 理解操作系统原理背后的人文背景与历史动机。
- 运用操作系统知识来分析和解决问题。
- 掌握操作系统设计的原理，为以后设计操作系统打下基础。

这里需要提醒的是，本书阐述的是操作系统的原理，它不依赖于任何具体的实现，而是凌驾于所有具体商业操作系统的进程实现之上。即本书所阐述的思想和原理对所有操作系统都适用。但具体商业操作系统在应用这些原理时可以有很灵活的方式。事实上，具体的商业操作系统在应用这些原理时确实采取了不同的方式，有的更为精密，也有的偷工减料。另外，由于我们注重的是原理，对个体机制实现时采取的数据结构通常不作琐细的论述，而是点到为止。这是因为数据结构必须以真正的操作系统为蓝本进行讲解，而真正的商用操作系统使用的数据结构通常非常复杂，对此进行繁琐的讲解将把学生弄得晕头转向，而不利于对操作系统核心原理的把握。

当然了，如果要达到能够设计开发真正商业操作系统的境界，读者还需要进行“操作系统工程”或“操作系统实现”的学习。而这种工程或实现的课程通常以具体的操作系统为对象进行讲述。这些具体的操作系统可以是 Windows、UNIX、Linux，当然也可以是其他一些非主流商业操作系统。如果能够将本书阐述的原理与操作系统工程或实现相结合，将取得更好的效果。

最后，本作者感谢下列人士为本书审阅书稿：上海交通大学的陈凌峰、张漳、顾夏中、徐燕和美国密歇根大学的鞠晓恩，其中陈凌峰和张漳将本人的讲课做了原始记录。

现在就让我们一起来揭示秘密，数清操作系统里的星星吧。

目 录 CONTENTS

前言

第一篇 基础原理篇

第1章 操作系统导论	2
引子: 智者的挑战	2
1.1 人造学科	3
1.2 程序是如何运行的	5
1.3 什么是操作系统	7
1.4 魔幻与管理	8
1.5 用户程序与操作系统	9
1.6 操作系统的范畴	11
1.7 为什么学习操作系统	12
思考题	13
第2章 操作系统历史	14
引子: 操作系统进化的推动因素	14
2.1 第一阶段: 状态机操作系统 (1940 年以前)	15
2.2 第二阶段: 单一操作员、单一控制 端操作系统 (20 世纪 40 年代) ...	16
2.3 第三阶段: 批处理操作系统 (20 世纪 50 年代)	16
2.4 第四代: 多道批处理操作系统 (20 世纪 60 年代)	18
2.5 第五代之一: 分时操作系统 (20 世纪 70 年代)	19
2.6 第五代之二: 实时操作系统	20
2.7 第六代: 现代操作系统 (1980 年以后)	21
2.8 操作系统的演变过程	22

2.9 操作系统的未来发展趋势	25
思考题	26
第3章 操作系统基本概念	27
引子: “差不多”精神	27
3.1 计算机硬件基本知识	28
3.2 抽象	31
3.3 内核态和用户态	31
3.4 操作系统结构	33
3.5 进程、内存和文件	35
3.6 系统调用	36
3.7 壳	37
思考题	39

第二篇 进程原理篇

第4章 进程	42
引子	42
4.1 进程概论	43
4.2 进程模型	44
4.3 多道编程的效率	44
4.4 进程的产生与消失	46
4.5 进程的层次结构	47
4.6 进程的状态	47
4.7 进程创立	49
4.8 进程与地址空间	49
4.9 进程管理	50
4.10 进程的缺陷	52
思考题	52
第5章 线程	53
引子	53
5.1 进程的分身术——线程	54
5.2 线程管理	55

11.6 多道编程的内存管理	144
11.7 闲置空间管理	150
思考题	152
第12章 页式内存管理	153
引子	153
12.1 基址极限管理模式的问题	154
12.2 分页内存管理	156
12.3 分页系统的优缺点	160
12.4 翻译速度	161
12.5 缺页中断处理	163
12.6 锁住页面	163
12.7 页面尺寸	164
12.8 内存抖动	165
思考题	167
第13章 页面更换算法	168
引子	168
13.1 页面需要更换	169
13.2 页面更换的目标	169
13.3 随机更换算法	170
13.4 先进先出算法	170
13.5 第二次机会算法	171
13.6 时钟算法	172
13.7 最优更换算法	172
13.8 NRU 算法	173
13.9 LRU 算法	174
13.10 工作集算法	179
13.11 工作集时钟算法	181
13.12 页面替换策略	181
思考题	182
第14章 段式内存管理	184
引子	184
14.1 分页系统的缺点	185
14.2 分段管理系统	186
14.3 分段的优缺点	188
14.4 段页式内存管理	189
14.5 段号是否占用寻址字位	190
14.6 讨论: 否定之否定的嵌套——纯粹 分段与逻辑分段、分页与段页	191
思考题	192

第四篇 文件原理篇

第15章 磁盘操作	194
引子	194
15.1 磁盘组织与管理	195
15.2 磁盘的结构	195
15.3 盘面的结构	196
15.4 磁盘驱动器的访问速度	197
15.5 操作系统界面	198
15.6 磁盘调度算法	199
思考题	202
第16章 文件基础	203
引子	203
16.1 为什么需要文件系统	203
16.2 什么是文件系统	204
16.3 文件系统的目标	205
16.4 文件的基本知识	205
16.5 从用户角度看文件系统	205
16.6 地址独立的实现机制: 文件夹	212
16.7 文件系统调用	214
16.8 内存映射的文件访问	215
思考题	216
第17章 文件系统实现	217
引子	217
17.1 文件系统的布局	218
17.2 文件的实现	219
17.3 目录实现: 地址独立的实现	226
17.4 闲置空间管理	231
思考题	232
第18章 文件系统	233
引子	233
18.1 文件系统访问控制	234
18.2 主动控制: 访问控制表	235
18.3 能力表	236
18.4 访问控制的实施	238
18.5 文件系统性能	239
18.6 提高系统性能的方法	245
18.7 文件系统设计分析: 日志结构的文件系统	248

18.8 海量数据文件系统	250
思考题	251

第五篇 I/O 原理篇

第 19 章 输入输出	254
引子	254
19.1 什么是输入输出	255
19.2 输入输出的目的	256
19.3 输入输出硬件	256
19.4 输入输出软件	262
19.5 I/O 软件分层	266
思考题	269

第六篇 多核原理篇

第 20 章 多核结构与内存	272
引子	272
20.1 以量取胜	273
20.2 多核基本概念	273
20.3 多核的内存结构	277
20.4 对称多处理器计算机的启动 过程	279
20.5 多处理器之间的通信	279
20.6 SMP 缓存一致性	281
20.7 多处理器、超线程和多核的 比较	281
思考题	282
第 21 章 多核环境下的进程同步 与调度	283
引子	283
21.1 多核环境下操作系统的修正	284
21.2 多核环境下的进程同步与调度	284
21.3 多核进程同步	284
21.4 硬件原子操作	285
21.5 总线锁	285
21.6 多核环境下的软件同步原语	286

21.7 旋锁	286
21.8 其他同步原语	289
21.9 多核环境下的进程调度	289
21.10 多核环境下的能耗管理	292
21.11 讨论：多核系统的性能	293
思考题	295

第七篇 操作系统设计原理篇

第 22 章 操作系统设计之原理	298
引子	298
22.1 操作系统设计的追求	300
22.2 操作系统设计的第 1 条哲学原理： 层次架构	300
22.3 操作系统设计的第 2 条哲学原理： 没有对错	301
22.4 操作系统设计的第 3 条哲学原理： 懒人哲学	302
22.5 操作系统设计的第 4 条哲学原理： 让困于人	303
22.6 操作系统设计的第 5 条哲学原理： 留有余地	304
22.7 操作系统设计的第 6 条哲学原理： 子虚乌有——海市蜃楼之美	305
22.8 操作系统设计的第 7 条哲学原理： 时空转换——沧海桑田之变	305
22.9 操作系统设计的第 8 条哲学原理： 策机分离与权利分离	305
22.10 操作系统设计的第 9 条哲学原理： 简单为美——求于至简、 归于永恒	306
22.11 操作系统设计的第 10 条哲学原理： 适可而止	306
思考题	307
结语	308
参考文献	310

PART ONE

第一篇 基础原理篇

对于任何一门课程来说，首要探讨的问题就是这门课的主题到底是什么？对于刚接触操作系统的入门者来说，自然想到的问题当然也会是操作系统到底是什么东西。回答这个问题是本篇的职责。此外，操作系统作为计算机的核心控制系统，它在计算机运行过程中扮演什么角色？它的来历是什么？它有一些什么基本概念？我们应该如何看待操作系统？它是如何参与到程序的执行过程中的？这些也是学习操作系统需要了解的基本问题。

本书的基础原理篇就是针对上述问题而写成。它对这些问题进行解答和讨论，并为我们接下来介绍操作系统的核心功能部件打下基础和铺垫。本篇包含第1章至第3章内容。第1章的内容包括智者的挑战、人造与神造、程序是如何运行的、什么是操作系统、魔术与管理、用户程序与操作系统、操作系统范畴和为什么学习操作系统。第2章探讨操作系统演变的主要过程：从单一控制终端单一操作员，到批处理、多道批处理、分时操作系统、实时操作系统、现代操作系统；对商业操作系统演变的过程进行分析，然后探讨操作系统分类和操作系统的未来发展趋势。第3章简要回顾计算机硬件基本知识，探讨什么是“抽象”，讲解用户态与内核态，阐述操作系统结构、系统调用、操作系统的壳等知识。

本篇最为重要的核心思想是操作系统在计算机运行过程中扮演的角色：魔术师和管理者。魔术师将丑陋变得美好，将没有变为有，将少变为多；而管理者则对所有计算机资源进行管理以达到公平和效率的“双料”境界。对操作系统这两个角色的理解将非常有助于对进程、线程、虚拟内存、文件系统和输入输出系统的掌握。



化丑为美：魔术师是操作系统扮演的一个根本角色

第1章 操作系统导论

引子：智者的挑战

西方有一个著名的故事，名曰：智者的挑战。相传很久以前，有座村子里住着一位智者。同村有个年轻人学到一些知识后就想来挑战这位智者。于是，年轻人想到了一个方法。他来到智者面前，将双手放在背后，问这个智者：

“我刚刚从树上抓了只鸟，现在在我手上。你能告诉我，这只鸟是活的还是死的呢？”

这是一个诡诈的问题。因为，如果回答“是活的”，则这个年轻人只需要在背后将鸟掐死，然后给智者看这只死鸟；如果回答“是死的”，则这个年轻人只需要将鸟放飞即可。这样，无论智者如何回答，年轻人都可以让智者答错，然后可以大大地嘲笑一番智者的水平。

作为智者，当然一眼看出了年轻人的诡计。但是又不能不回答这个问题。因为不回答问题等于承认答不上来，当然也就不是什么智者了。但如何回答呢？

智者的回答简洁、精妙，甚至妙不可言：“As you will”。

乍看上去，这个答案没有什么神奇之处。但如果读者的英语水平很高，就可以看出其中的奥妙。will 这个词在英语中的意思是意愿或意志。因此这个答案的意思是：这只鸟的死活与年轻人的意志保持一致：年轻人的意志是让鸟活，这只鸟就是活的；年轻人的意志是让鸟死，则这只鸟就是死的。

如果到此止步，则这个答案存在着巨大的漏洞：年轻人可以将鸟掐死，但坚持说自己的意志是让鸟活着，即自己不小心将鸟掐死了，而自己的意志却是想让鸟活着。这样的话，智者的回答就错了。

妙不可言的是，will 在英语中还有一层意思：将要，即将要发生的事或将要采取的行动。就是说，这只鸟的死活与年轻人将要采取的行动保持一致：年轻人将要放飞这只鸟，这只鸟就是活的；年轻人将要掐死这只鸟，这只鸟就是死的。

因此，智者的回答将人的意志与行为全部包括进来。这样，即使年轻人声称他的意愿和行

为并不一致，智者的回答也正确无误。

好，我们知道了智者的回答。但这与操作系统课程有什么关系吗？

有！很多人都觉得操作系统枯燥、乏味，甚至令人厌烦。更有人说懂不懂操作系统没有关系。不是很多人在学习操作系统之前就已经写过程序了吗？有的人甚至已经写过很大很复杂的程序了。可见，不懂操作系统并不妨碍我们学习使用计算机。

如果读者这样想，我劝你再想一想。你虽然写过程序，可你知道程序到底是如何在计算机上运行的吗？如果不知道，你怎么敢肯定你的程序总是会运行正确呢？你怎么敢说你写的程序最大限度地利用了系统的能力了呢？一个人觉得操作系统没用，那是因为他不知道怎么用，或者他没有用操作系统的意愿。说明白一点，你如果认为操作系统没有用，那是因为你的编程和程序开发处在一个低级的水平上。如果你掌握了操作系统，你的编程水平将显著提高。

换句话说，操作系统有没有用，我的回答是“*As you will*”。你如果有意愿或者有行动使用操作系统，操作系统就是有用的；如果你没有意愿或行动，则操作系统就是没有用的。当然了，我希望读者在看完这本书后能够领悟到操作系统的巨大用途。万一在读完本书后，读者还是困惑或者觉得没用，我唯一能说的也是“*As you will*”。当然，我希望这种情况发生的概率不大。

1.1 人造学科

要想学好操作系统，具有恰当的思维模式是十分必要的。这个思维模式就是本书所强调的“哲学”：一种思维方式或一种生活方式。我们以一个问题来说明这一点。这个问题是：什么是计算机的根本特征？

对于这个问题，相信很多人会说计算机就是个计算机器，或者是用来进行大规模计算的机器，或者是用于数据处理的机器，或者是具备某些其他具体功能的机器。这些回答当然没有错，问题是这些答案并不能帮助我们更好地学习理解计算机。就像我们问“张三这个人的根本特征是什么”，而回答是“张三有175cm高”一样。这种答案虽然是正确的，但意义不大，因为我们无法从答案中推导出一系列有用的结论。

那这个问题该如何回答呢？这就要看我们对事物的观察程度。如果我们仔细看看身边的事物，就会发现所有的东西可以划分为两类：一类是本来就存在于自然中，人类所做的只不过是发现；另一类是本来并不存在，人类所做的是发明。第一类事物我们称之为神造事物或者自然存在的事物，第二类事物当然就是人造事物。从这个思维模式上看，计算机毫无疑问就是人造事物，这正是我们所需要的答案，即计算机的根本特征是“人造”。

引申一下可知，计算机学科就是一个人造学科。那么知道计算机学科是人造学科对我们学习计算机有什么帮助呢？有，太多了。下面我们来看看人造和神造有什么区别。

人造学科四个特点：

- 不精确、具有相对性。
- 从对人类活动的观察导出。
- 依赖于人的主观判断力。
- 通常符合人的直觉。

第一个特点就是所谓的“没有对错”。在人造的学科里，没有什么绝对的对或者错，而只有所谓的“好”或者“坏”，“有意义”或者“没意义”。例如，如果本书在讨论计算机时某个方面的论述与你见到的计算机不一样，这不说明本书错了。就算世界上没有本书所论述的计算机存在，也不说明本书错了。我们只需要按照本书的论述再造一台计算机即可。但是，本书论述的计算机与你知道的计算机之间可以进行好和坏的比较。

第二个特点说的是人造学科是从什么得到灵感的，那就是“对人类活动的观察”。这样，读者只要对人类生活仔细观察，就可以很容易地理解计算机里面的许多原理。例如，在操作系统中，广泛使用的栈和队列就是对从生活中观察到的现象进行抽象所获得，如图 1-1 所示。

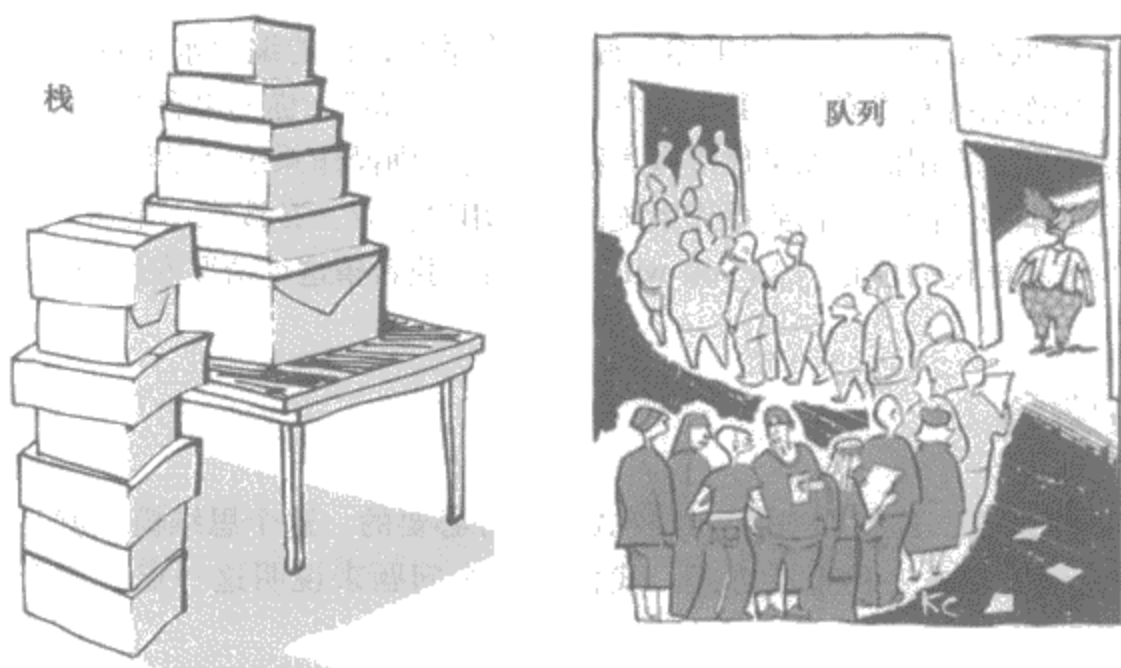


图 1-1 从观察人类活动而获得的栈和队列结构

第三个特点说的是在人造学科里，人的主观能动性起着关键的作用。不同的人观察同样的现象，得出的结论或抽象出的东西可能不一样，甚至完全相反。这样，多数人所认同的抽象将成为人造学科里的标准，即存在少数服从多数的原则。

第四个特点说的是人造学科里面的许多原理与人的直觉直接呼应，即如果我们按照人的直觉去理解这些原理，就会十分直截了当。例如，操作系统里面的同步机制与人类男女谈恋爱时所用的约会机制十分相似。对于一个谈过恋爱或与别人约会过的人来说，如果将自己谈恋爱的直觉用在操作系统进程的学习上，就会发现进程同步是个十分容易理解的概念。

相对于人造学科，神造学科刚好具有相反四个属性：

- 精确、绝对。
- 从对自然存在的观察导出。
- 不依赖于人的主观判断力。
- 通常违反人的直觉。

第一个特点说的是神造的事物具有精确、绝对的属性。对于这种学科，存在正确与错误之分，我们提出的观点要么正确，要么错误，不存在中间状态。例如，纯数学领域的各种运算，如 2^2 的结果应该是 4。如果运算的结果不是 4，则属于运算错误。

第二个特点说的是人类对这些事物的理解是从对自然存在的观察中获得。例如，牛顿通过观察苹果落地的自然现象和严密的推理，得出了万有引力定律。

第三个特点说的是这些观察的结果是不依赖于人的主观能动性的。如果一个人的观察结果是正确的话，那么他的观察结果将和所有正确的观察结果一样，而绝不会是两样。从另外一个角度说，一个人的观察抽象结果是可以被他人验证的。例如， $30\,000\,000\,000 + 20\,000\,000\,000$ 对于任何人来说，如果计算正确，则结果必然是 $50\,000\,000\,000$ 。

第四个特点说的是如果我们按照人的直觉来学习，就会面临重重困难。因为人的思维与神不一样。自然，按照人的思维模式将很难理解神所创造的这一切。这就是为什么在这些学科耕耘的人都必须依赖灵感的出现，和严密、一丝不苟的数学与逻辑推理（见图 1-2）。



图 1-2 从观察自然存在和严密的数学推理获得的质能方程

明白了计算机是人造事物，操作系统是一个人造的系统，我们就可以按照人造物的特点来进行学习，从而易如反掌地掌握操作系统的原理。

1.2 程序是如何运行的

计算机程序是怎样运行的呢？对于多数人来说，或多或少地知道任何程序必须首先有人写出来，即“编程”，然后放到计算机里即可运行。这种解释当然是过于简单了。计算机程序的运行实际上是一件十分复杂的事情，牵扯到方方面面。

首先当然是进行编程，而编程需要计算机程序设计语言作为基础。对于绝大多数编写程序的人来说，使用的编程语言称为“高级程序设计语言”，如 C，C++，Java 等。但由于计算机并不认识高级语言编写的程序，编好的程序需要通过编译变成计算机能够识别的机器语言程序，而这需要编译器和汇编器的帮助。其次，机器语言程序需要加载到内存，形成一个运动中的程序，即“进程”，而这需要操作系统的帮助。进程需要在计算机芯片 CPU 上执行才算是真正在执行，而将进程调度到 CPU 上运行也由操作系统完成。再次，在 CPU 上执行的机器语言指令需要变成能够在一个个时钟脉冲里执行的基本操作，这需要指令集结构和计算机硬件的支持，而整个程序的执行过程还需要操作系统提供的服务和程序语言提供的执行环境（runtime environment）。这样，一个从程序到微指令执行的整个过程就完成了。图 1-3 所示的就是这个过程。

当然了，图 1-3 描述的从程序到结果的演变过程还是过于简单。我们只是从一个线性的角度来看程序的演变过程，而没有考虑到各种因素之间的穿插和交互过程。不过，对于才入门的计算机专业学生来说，这种描述能够帮助理解整个程序是如何在计算机上执行的问题。

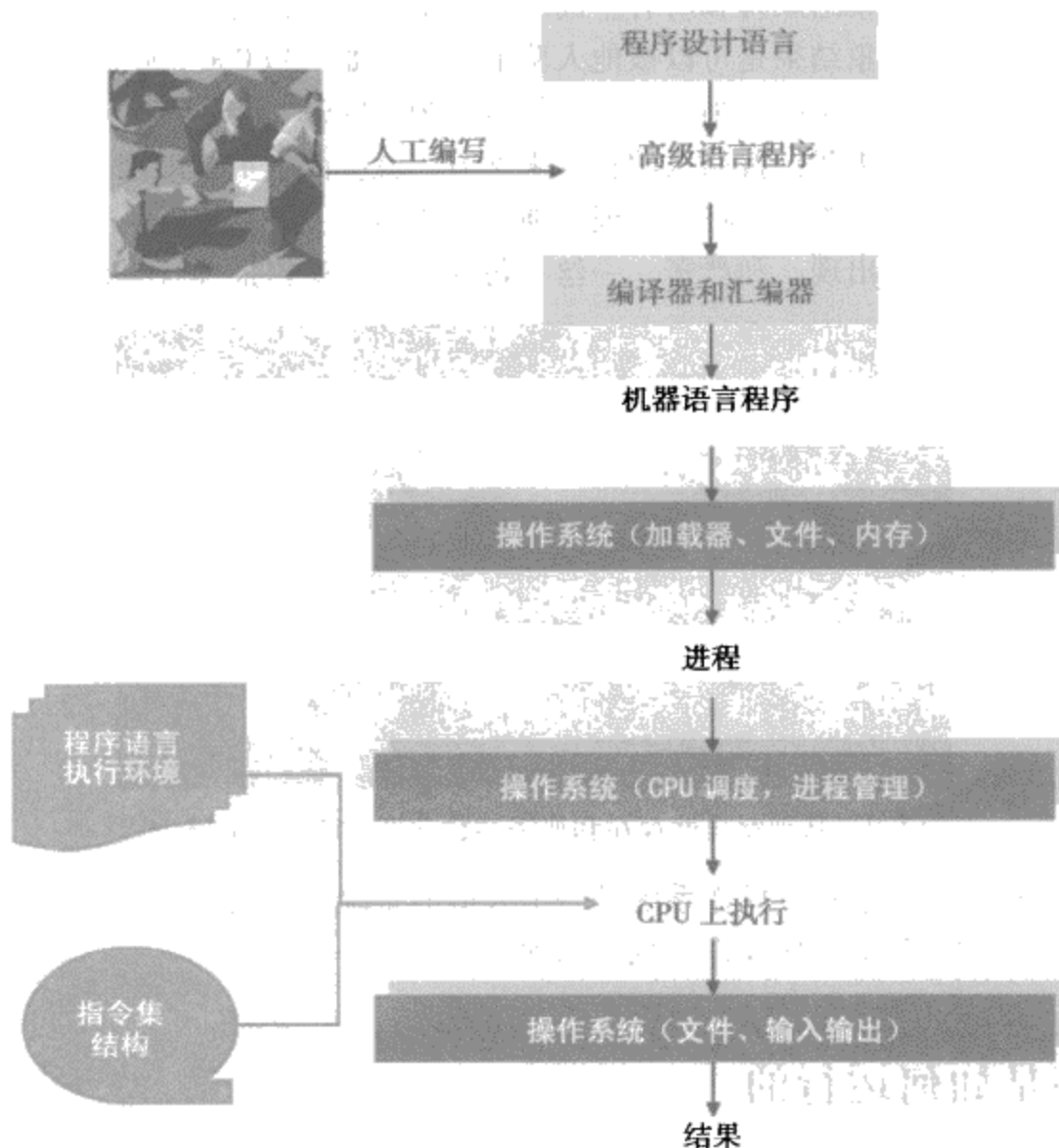


图 1-3 由程序到结果的演变

从这个描述可以看出：程序的运行至少需要如下 4 个因素：

- 程序设计语言。
- 编译系统。
- 操作系统。
- 指令集结构（计算机硬件系统）。

这 4 个因素都将是大学学习的专业课程。需要注意的是，操作系统在程序的执行过程中具有关键的作用，本书要做的就是阐述这个关键作用是如何发挥的。

需要提醒的是，图 1-3 给出的程序执行过程是从高级语言编写的程序开始，而实际并非总是这样。事实上，程序可以直接在机器语言或汇编语言上编写。用这种称为“低级”的语言编写出来的机器语言程序无需经过编译器的翻译就可以在计算机指令集上执行。如果是在汇编语言上编写的汇编程序，则只需要经过汇编器的翻译即可加载执行。

1.3 什么是操作系统

操作系统这个术语听上去稀松平常，并不给人任何兴奋的感觉，甚至有点俗气。原因在于中文的“操作”这个词：提到操作员，通常让人想起操作车床、磨床和起重机的穿着油腻工作服的工人，自然让人兴奋不起来。将英文的 Operating 翻译为中文的“操作”，是因为翻译的人没有真正理解英文 Operating Systems（操作系统的英文名称，一般缩写为 OS）这个名字所蕴含的精髓。

那么英文的 Operating Systems 意味着什么呢？

各位见过手术过程吗？在手术室里，主刀大夫称为 Operating Surgeon。在整个手术过程中，主刀大夫具有至高无上的权威：他说要打麻药，麻醉师就要赶紧打麻药；他说需要手术钳，助理大夫就赶忙递给他手术钳；他说需要止血，护士就得马上拿止血药棉来止血。整个手术最关键的部分，切开皮肤、拿掉器官、安装移植器官等均由主刀大夫完成。当然，主刀大夫有时候也会将某些任务，如缝合创口，交给助理大夫或护士来做，但整个手术的过程皆由其主控。一句话，Operating Surgeon 就是掌控整个手术过程、具有精湛技术和敏锐判断力的医师。

引申至非医学领域，Operating Person 意思是操刀手，就是掌控事情的人。再将 Person 这个词换成 System，则 Operating Systems 指的就是掌控局势的一种系统。也就是说计算机里面的一切事情均由 Operating Systems 来掌控。那么，我们现在面临两个问题：第一个问题是操作系统到底是什么？第二个问题是操作系统到底操控什么事情？

我们先回答第一个问题。既然操作系统是掌控计算机局势的一个系统，自然很重要。但这个说法并不能帮助读者理解操作系统，也无法形成有形的概念。如果我们换个说法：操作系统是一个介于计算机和应用软件之间的一个软件系统，则概念就具体多了。从这个定义出发，我们知道操作系统的上面和下面都有别的对象存在：下面是硬件平台，上面是应用软件，如图 1-4 所示。

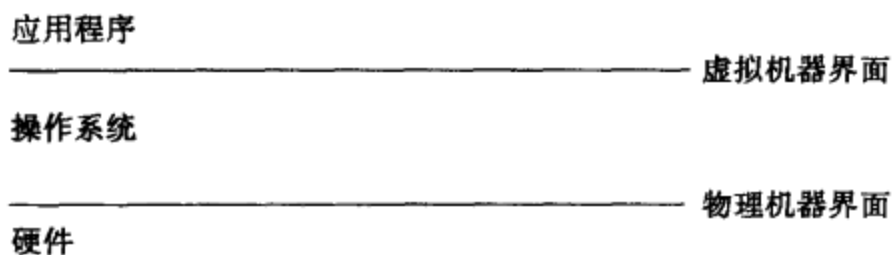


图 1-4 操作系统上下界面

再来回答第二个问题。我们现在知道操作系统代表的是掌控事情的系统。掌控什么事情呢？当然是计算机上或计算机里发生的一切事情。最原始的计算机并没有操作系统，而是直接由人来掌控事情，即所谓的单一控制终端、单一操作员模式。但是随着计算机复杂性的增长，人已经不能胜任直接掌控计算机了。于是我们编写出操作系统这个“软件”来掌控计算机，将人类从日益复杂的任务中解脱出来。这个“掌控”有着多层深远的意义。

首先，由于计算机的功能和复杂性不断发生变化（趋向更加复杂），操作系统所掌控的事情也就越来越多，越来越复杂。同时，操作系统本身能够使用的资源也不断增多（如内存容

量)。这是早期驱动操作系统不断改善的根本原因。

其次,既然操作系统是专门掌控计算机的,那么计算机上发生的所有事情自然需要操作系统的知晓和许可,未经操作系统同意的任何事情均视为非法事情,也就是病毒和入侵攻击所试图运作的事情。作为操作系统的设计人员,我们当然要确保计算机不发生任何我们不知情或不同意的事情。但是人的能力是有限的,人的思维也是有缺陷的,我们设计出的系统自然不会十全十美,也会有缺陷的,这就给了攻击者可乘之机。操作系统设计人员和攻击者之间的博弈是当前驱动操作系统改善的一个重要动力。

再次,掌控事情的水平有高低之分,有效率不同之分。就像手术大夫之间也有水平高低之分。为了更好地掌控事情,为了更好地满足人类永不满足的各种越来越苛刻的要求,操作系统自然需要不断改善。这种改善在过去、现在和将来都会继续下去。

好了,说到这里,我们可以给操作系统做一个定义了:操作系统是一个软件系统,使计算机变得好用(将人类从繁琐、复杂的对机器掌控的任务中解脱),使计算机运作变得有序(操作系统掌控计算机上所有事情)。

总结起来就是:操作系统是掌控计算机上所有事情的软件系统。

从这个定义可以引申出操作系统的功能包括:

- 替用户及其应用管理计算机上的软硬件资源。
- 保证计算机资源的公平竞争和使用。
- 防止对计算机资源的非法侵占和使用。
- 保证操作系统自身正常运转。

1.4 魔幻与管理

将上面所陈述的操作系统功能进行提升,就可以得出操作系统所扮演的两个根本角色是:管理者和魔幻家。只要记住了这两个角色,就差不多明白什么是操作系统。

1.4.1 魔幻家角色

将计算机以一个更加容易、更加方便、更加强大的方式呈献给用户使用。直白地说,就是把差的东西变好,把少的东西变多,把复杂的东西变得容易。例如,如果在裸机上直接编程是很困难的,因为各种数据转移均需要用户自己来控制,对不同设备要用不同命令来驱动,而这对一般人来说很难胜任。操作系统将这些工作从用户手中接过来,从而让用户感觉到编程是一件容易的事(相对来说,编程对于有些人来说永远很难)(见图1-5)。

操作系统通过进程抽象让每一个用户感觉到有一台自己独享的CPU;通过虚拟内存的抽象,让用户感觉到物理

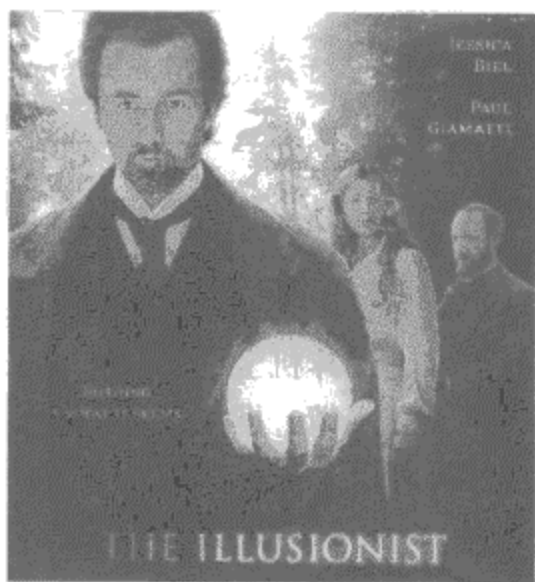


图1-5 操作系统就是一个魔术师

内存空间具有无限扩张性。这就是把少变多。当然，操作系统的把少变多不是无中生有。变多也不是无限多，只是针对磁盘容量的大小。

1.4.2 管理者角色

管理计算机上软硬件资源。例如，操作系统对 CPU、内存、磁盘等的管理，使得不同用户之间或者同一用户的不同程序之间可以安全有序的共享这些硬件资源。那怎么让用户很好地利用这些硬件资源呢？就是分块（parcels out），把硬件分块出来给应用程序使用。这里关键的原则是有效和公平，这是管理者的必备素质。有效指的是不能浪费资源，公平指的是每个人都有享有资源的可能，即不能有不公平的现象。当然真正的公平是并没有的事，这很像人类生活的现实。不过追求公平却是我们的本能，在虚拟世界里尽可能公平一点还是非常应该的，至少应该是操作系统设计时的不懈追求。

根据管理的资源不同，操作系统具体功能包括：

- CPU 管理，即如何分配 CPU 给不同应用和用户。
- 内存管理，即如何分配内存给不同应用和用户。
- 外存管理，即如何分配外存（磁盘）给不同应用和用户。
- I/O 管理，即如何分配输入输出设备给应用和用户。

除了对上述资源进行管理和抽象外，操作系统作为掌控一切的软件系统，其自身必须是稳定和安全的，即操作系统自己不能出现故障。因此，操作系统本身的设计还需包括如下两项：

- 健壮性管理，即如何确保操作系统自身的正常运作。
- 安全性管理，即如何防止非法操作和入侵。

为完成上述所列的功能，操作系统设计人员构思了许多机制。而所有这些机制均有其来龙去脉，其背后隐含的是人的哲学思维。我们这门课就是要讲解操作系统后面的哲学原理，并依据这些哲学原理阐述操作系统是通过何种机制、以何种方式完成上述列举的各种管理功能。

1.5 用户程序与操作系统

前面说过，操作系统上下分别是虚拟机器界面和物理机器界面。处于物理机器下面的是硬件，而硬件和操作系统的关系将是本书的关注点。处于虚拟机器界面上面的是应用软件，应用软件和操作系统的关系不是本书的重点，而是系统编程或底层编程等课程的关注点。在这里，我们只想简要讨论一下应用程序和操作系统的关系，因为这个关系对理解操作系统非常重要。

那么，操作系统和应用程序之间是个什么关系呢？很显然，操作系统为用户程序提供了一个虚拟机器界面，而应用程序运行在这个界面之上。但这个答案似乎太抽象。并不能帮助深入理解它们之间的关系。我们前面讲过，操作系统是一个程序，而用户程序也是程序，程序和程序之间能有什么关系呢？无非是调用和被调用的关系。

那操作系统和用户程序之间到底谁是调用者，谁是被调用者呢？答案似乎很清楚。操作系统通过虚拟机器界面给用户程序提供各种服务，用户程序在运行过程中不断使用操作系统提供的服

务来完成自己的任务。例如，用户程序在运行过程中需要读写磁盘，这个时候就需要调用操作系统的服务来完成磁盘读写操作。如果需要收发数据包，也需要调用操作系统的服务来完成。当调用这些服务时，控制从用户程序转移到操作系统，而操作系统在完成这些服务后将控制返回给用户程序。在这种思维模式下，用户程序是主程序，操作系统是子程序，如图 1-6 所示。

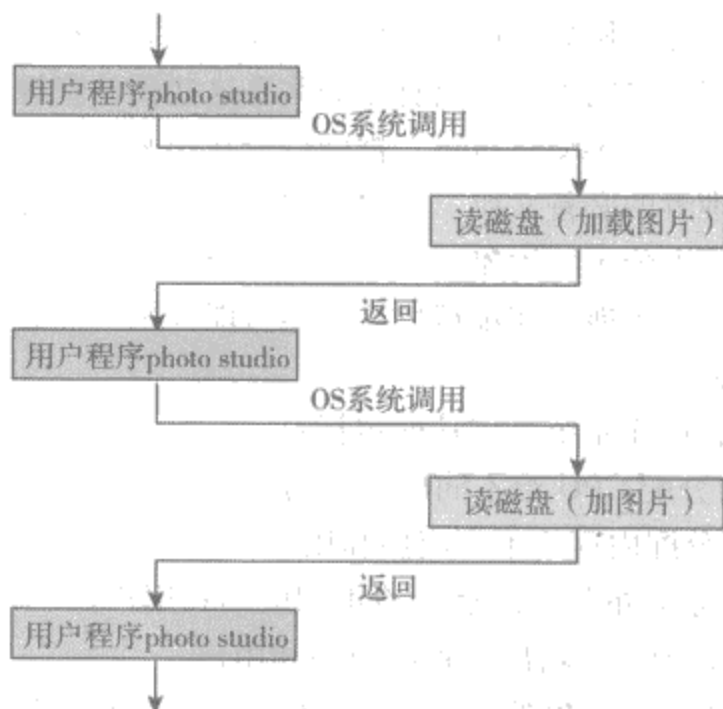


图 1-6 用户程序为主程序，操作系统为子程序

但是有正就有反，这就是哲学中的矛盾论。如果我们从另一个角度来看，会得出相反的结论。系统启动之后最先启动的是什么程序？操作系统。用户程序不能在操作系统启用之前启动（除非是很厉害的病毒）。在此之后，每次启动一个用户程序，都相当于操作系统将控制转移给用户程序；而在用户程序执行完毕后，控制又回到操作系统。这样看上去，操作系统是主程序，它在一生当中不断调用各种应用程序，而每个应用程序执行完之后再回到操作系统。就这样循环往复，直到无穷或机器关闭。在此种思维模式下，操作系统是主程序，用户程序是子程序，如图 1-7 所示。

上述两种看法完全相反，但又似乎都有道理，有谁对谁错之分吗？没有。我们说过，人造学科没有对错之分，只有好坏之分。你喜欢哪个观点你就持那种观点，哪个观点帮助你理解操作系统，你就持哪种观点。如果两种观点都有帮助，你可同时持有两种观点。

当然了，上述关系描述是非常简单化的。实际上，操作系统和各种用户程序可以看作是互相

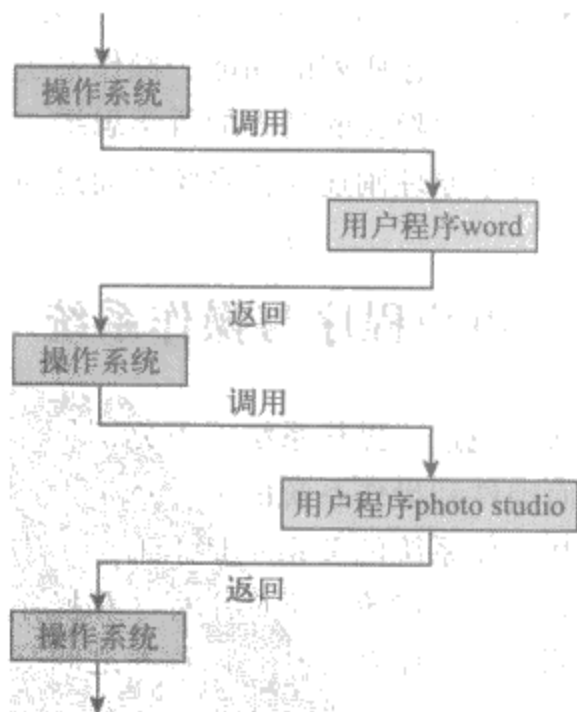


图 1-7 操作系统为主程序，用户程序为子程序

调用,从而形成一个非常复杂的动态关系。了解并阐述这种复杂的动态关系就是本书的目的。

1.6 操作系统的范畴

我们前面讲过了操作系统的两个角色:魔术师和管理者。这两个角色之间既有区别,又有联系。为了完成不同的任务,操作系统有时需要扮演魔术师的角色,有时需要扮演管理者的角色,有时则需要同时扮演这两个角色。那操作系统要完成的任务具体有哪些呢?前面提到过:

- CPU 管理,即如何分配 CPU 给不同应用和用户。
- 内存管理,即如何分配内存给不同应用和用户。
- 外存管理,即如何分配外存(磁盘)给不同应用和用户。
- I/O 管理,即如何分配输入输出设备给应用和用户。

CPU 管理就是我们将要讲的进程管理。进程管理的主要目的有三个:第一个是公平,即每个程序都有机会使用到 CPU。第二个是非阻塞,即任何程序不能无休止地阻挠其他程序的正常推进。如果一个程序在运行过程中需要输入输出或者别的什么事情而发生阻塞,这个阻塞不能妨碍别的进程继续前进。就像人类世界,缺了谁地球都照样旋转。第三是优先级。在人类生活中人的地位不是完全一样的,地位高的就比你优先级高。人类把自己生活中的这种关系搬到操作系统里面,就有了优先级的概念,即某些程序比另外一些程序优先级高。如果优先级高的程序开始运行,则优先级低的程序就要让出资源。就像我们经常说的,我们坚决反对大锅饭,应该让一部分人(程序)先富起来。

内存管理主要是管理缓存、主存、磁盘、磁带等存储介质所形成的内存架构。为此目的,操作系统设计人员发明了虚拟内存的概念,即将物理内存(缓存和主存)扩充到外部存储介质(磁盘、光盘和磁带)上。这样内存的空间就大大的增加了,能够运行的程序的大小也大大的增加了。内存管理的另一个目的是让很多程序共享同一个物理内存。这就需要对物理内存进行分割和保护,不让一个程序访问另一个程序所占的内存空间,专业术语称为运行时不能越界。在生活中,就是我家的东西不希望你跑来拿。

存储管理就是众所周知的文件系统了。文件系统的主要目的是将磁盘变成一个很容易使用的存储媒介提供给用户使用。这样我们在访问磁盘时无需了解磁盘的物理属性或数据在磁盘上的精确位置,诸如磁道、磁柱、扇面等。当然,文件系统还可以建立在光盘和磁带上。只是使用最为频繁的文件系统都以磁盘为介质。

设备管理就是管理输入输出设备。其目的有两个:一是屏蔽不同设备的差异性,即用户用同样的方式访问不同的设备,从而降低编程的难度;二是提供并发访问,即将那些看上去并不具备共享特性的设备,如打印机,变得可以共享。

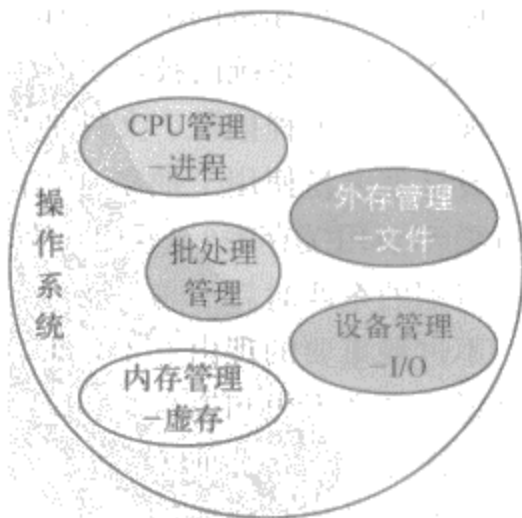


图 1-8 操作系统的 5 个核心构件

另外还有一个任务称为批处理。批处理提供一种无需人机交互的程序运行模式。有时我们不需要人来交互，就批处理交给计算机。主要是要达到吞吐量最大化，单位时间完成的任务最多。图 1-8 描述的是操作系统的 5 个核心功能。

当然，在真实的操作系统里，上述 5 个核心部件不一定界限分明，甚至它们不在同一个态势下运行（本书后面将说明这点）。图 1-9 描述的是 Windows 操作系统简化了的结构。

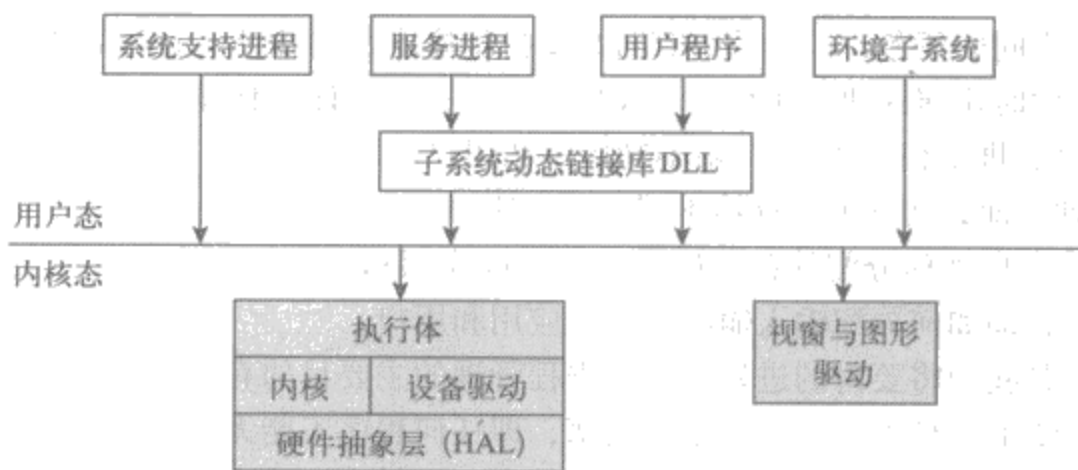


图 1-9 Windows 操作系统结构图

1.7 为什么学习操作系统

到目前，本书论述了操作系统是什么、操作系统的主要任务和操作系统与用户程序的关系后，读者应该体会到操作系统的重要性。但仅仅是因为操作系统重要就要学习它吗？世界上重要的东西太多了，难道我们都要学吗？即使是计算机专业的学生，不学操作系统也照样可以编程写软件。那我们为什么要学呢？当然我们有一千个理由要学，但这里仅给出几个。

首先，操作系统的功能在很多领域都使用。如果你做并发程序的开发：Web Service、分布式系统和网络，你会发现，这些领域大量使用了操作系统的概念和技术。如果你学好了操作系统，你就可以对你做的事情更加有信心。

其次，操作系统的技巧也在很多领域使用。如抽象、缓存、并发等。操作系统简单来说就是实现抽象：进程抽象、文件抽象、虚拟存储抽象等。而很多领域也使用抽象，如数据结构和程序设计就大量使用了抽象。记得抽象数据类型吗？记得抽象类吗？很多地方都用缓存。你做 Web 要不要用缓存呢？这些你都得做。如果学了操作系统，你就掌握了这些内容。触类旁通，你学习别的东西时就容易多了。

不过最重要的理由并不是上面两条，而是操作系统真的很有意思。对于一个计算机专业的人来说，难道不想知道自己写的程序到底是如何在计算机上运行的吗？读者一定见过汽车吧。汽车前面那个盖子叫前盖。很多人买车后第一件事是什么？打开前盖。那么打开前盖看到的是什么东西？马达，变速箱。为什么第一件事要打开前盖呢？因为好奇这辆汽车是怎么开动的（见图 1-10）。

那么对于一个程序设计员来说，有没有在看到一台计算机的时候，想过为什么计算机能进

行计算？有没有买来一台新计算机后就打开盖子呢？多数人恐怕没有打开过计算机外壳。不过，没有打开过也不用遗憾。因为即使你把计算机后盖打开，还是不能明白计算机是怎么运转的，此时只看到一堆硬件：芯片、主板、布线等，而这些硬件并不会告诉你太多有关计算机运转的信息。如果真的想知道计算机是怎么运转的，你就得学操作系统。当然，如果你想知道计算机在硬件层面上是如何运转的，则还应该学习计算机组成和体系结构等课程。

虽然学习操作系统很有趣，但并不是所有人都有这样的感觉。历史证明，对很多人来说学操作系统是一件很痛苦的事情。不过我希望阅读本书对读者来说是一件乐事，难道窥探奥秘不是一件激动人心的事吗？

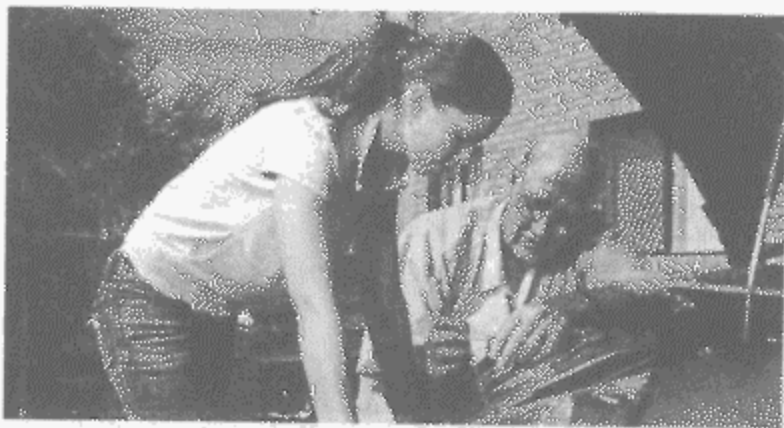


图 1-10 学习操作系统就是揭开覆盖在计算机上的“前盖”

思考题

1. 什么是操作系统？请用一句话描述你对操作系统的理解。
2. 你对操作系统和用户程序之间的关系有何看法？阐述你的视角。
3. 简要列出操作系统覆盖的范畴及每个范畴的核心内容。
4. 你为什么要学习操作系统？与本书列出的理由相同吗？简要阐述你的动机。
5. 操作系统要对不同的部件进行管理，请论述这些管理之间的异同点。
6. 设备管理要达到的目的是什么？
7. 有人说设备管理软件（设备驱动程序）因为经常由第三方提供，因此不应该作为操作系统的一部分。你对此有何看法？你认为应该如何判断一个软件是否属于操作系统？
8. 请列出程序执行过程中操作系统的介入情况。
9. 我们说操作系统是人造学科，根据是什么？
10. 人造学科的特点是什么？它对我们学习操作系统有何帮助？

第2章 操作系统历史

引子：操作系统进化的推动因素

上世纪末本世纪初，美国兴起了励志演讲（Motivational Speech）大潮，各种人等乐此不疲。在这拨励志大潮中，出现了多名影响力广泛的励志演说家。这些演说家所到之处，真是万人空巷，人潮涌动，场面之壮观令人叹为观止。

其中一位演讲家由于其名望很高接受了电视台采访。

记者：“您在励志演讲领域声名远播，影响巨大。您的每场演讲都人满为患，能否请您阐述一下什么是励志演说吗？”

励志专家：“你想知道吗？告诉你，我跟你一样，我压根就不知道什么是励志演讲。我只知道励志演讲非常流行，我只不过是利用这个潮流来赚钱而已。既然大家都喜欢，我就讲，至于我讲的东西是什么意思，我根本不知道。但是听众很喜欢！”

我想起了一句英语的歌谣：“sometimes when we touch, the honest is too much!”（有时，当我抚摸着你的时候，真实让我几乎无法承受！）。也许人们都不喜欢真实，于是就成就了很多人在论述操作系统历史（甚至人类历史）时的粉饰太平。

而操作系统的演变就是我们对计算机硬件进行粉饰的过程。

大多数的教材都有一章谈论相关领域的历史，操作系统自然也不例外。不过不同的教材谈论历史的目的却又不同。多数教材是为了铺垫一下本学科的发展背景，让学生了解相关领域里的发展大事，并无将历史与现实的发展联系起来。本课程谈论的历史则是以史明鉴，不是为了谈论历史而谈论历史，而是为了让学生明白操作系统为什么是现在这个样子，以及将来会是什么样子。从根本上把握操作系统这一计算机领域核心学科的脉搏，深刻理解社会变迁给计算机这门人造学科带来的不可抵挡的变化。同时，我们还将揭示计算机发展史上一些鲜为人知的重要细节，给学生一个窥探全貌的快感。

如我们前面所说，操作系统的不断发展与改善由两个因素驱动：

- 硬件成本的不断下降。

- 计算机的功能和复杂性的不断变化。

就是这些因素决定了操作系统的历史，我们一定要牢牢掌握这两个因素。硬件成本不断降低，就以硬盘为例，IBM 制造的第一张硬磁盘直径达 2 米，造价 100 多万美元，而容量仅仅只有 1MB。而现在一个容量 100GB 的硬盘成本只有几十美元。当然，过去的硬盘和今天的硬盘的制造技术完全不同，第 1 张硬盘的质量坚挺，可以当作咖啡桌来使用，而现在的硬盘（指盘片），非常软，根本不能承受重物。计算机复杂性的不断增加对于很多人来说并不感到惊奇。人类做的任何事情，都是越来越复杂。你住的地方不做整理很快就乱了，有没有这经验？当然，你的生活随着你年龄的增长，也会越来越复杂。最初，计算机的组件虽然巨大，但数量少，且功能简单。现在，一台计算机里面包括的元件数量实在是太多了。

硬件成本的下降和计算机复杂性的提高推动了操作系统的演变。成本降低意味着同样的价格可以买到更为先进的计算机。而复杂性提高自然需要操作系统的能力也得提高。就是这些变化使得操作系统从最初的仅仅几百或几千行代码的独立库函数，发展到今天的多达 4000 万行代码的 Windows XP 操作系统。而某些 Linux 的版本的代码行数更加庞大。

操作系统之所以越来越复杂是因为硬件质量和数量的提升使得操作管理的東西增多，而且人类永不满足的各种越来越苛刻的要求也使得计算机操作系统的复杂性增加。

另外，还有一个附加因素影响操作系统的发展，这就是操作系统和攻击者之间的博弈。这个世界上总有些人想利用计算机的缺陷来进行各种损人利己或损人又不利己的活动。操作系统在最初设计时根本就没有想到会有人从事破坏活动。大概因为最早的计算机工作者认为到达能够使用计算机的水平的人都是好人，无需设计任何安全机制。这样，在后来发现有人试图利用计算机进行不良操作时，就迫不得已修改操作系统，使其具有安全上的防范功能。每当操作系统改进了安全性，攻击者也会改良他们的攻击手段，这样循环往复，就造成操作系统安全水平和攻击者攻击水平不断交替上升的历史。

下面我们就来说一说操作系统是如何因上述驱动因素的变化而变化的。

2.1 第一阶段：状态机操作系统（1940 年以前）

这是计算机处在萌芽期时出现的操作系统。这种操作系统运行在英国人巴贝斯（Babbes）想像中的自动机中。所谓状态机操作系统实际上算不上是我们现在通常所定义的操作系统，而是一种简单的状态转换程序：根据特定输入和现在的特定状态进行状态转换而已。这个时候的计算机也不是现代意义上的计算机，而是所谓的自动机，其功能非常简单，可以用“原始”来形容。能做的计算也只限于加减法。这个时代的操作系统没有什么功能，不支持交互命令输入，也不支持自动程序设计。甚至这个时候还没有存储程序的概念。

驱动这一阶段操作系统的动力是个人英雄主义。因为此时尚无任何计算机工业、计算机研究及计算机用户。计算机及其操作系统的发展完全是某些人的个人努力。

这个阶段因为计算机刚刚出现，没有多少人能够接触到计算机，自然不存在什么安全问题。

这个阶段没有操作系统。如果非要说有的话，人就是这个时代的操作系统：因为自动机的

一切动作均是人在操控的。

2.2 第二阶段：单一操作员、单一控制端操作系统（20 世纪 40 年代）

这种单一操作员单一控制终端（SOSC, single operator, single console）的操作系统是在刚出现计算机时人们能想到的最直观的控制计算机的方式。这个时候的代表机型为美国宾夕法尼亚大学与其他机构合作制作的 ENIAC 计算机。这是第一台电子计算机，但不是第一台计算机。在这之前有个英国人造了一部机械计算机，通过手柄摇动能够进行计算。在 ENIAC 刚造出来的时候，谁都不知道计算机是怎么回事，所以没有操作系统的整体概念，唯一能想到的就是提供一些标准命令供用户使用，这些标准命令集合就构成我们的原始操作系统 SOSC。

SOSC 操作系统的设计目的就是满足基本的功能，并提供人机交互。在这种操作系统下，任何时候只能做一件事，即不支持并发和多道程序运行。操作系统本身只是一组标准库函数而已。操作系统并不自我运行，而是等待操作员输入命令再运行。用户想使用什么服务，就直接在命令行键入代表该服务的对应操作系统的库函数名（文件名）即可。这种操作系统的资源利用率很低：你输入一个命令就执行一个库函数，拨一下动一下。当操作员在思考时或进行输入输出时，计算机则静静的等待。当然了，从人的角度来看，效率并不低，你键入什么，计算机就立即执行什么。但从机器的角度考虑，因为时刻都等着人相对较慢的动作，效率就太低了。

由于这个时代的计算机很稀少，整个世界也只有几台，而人却不是，提高计算机的利用率就变得十分重要。

2.3 第三阶段：批处理操作系统（20 世纪 50 年代）

为了提高单一操作员单一控制终端的操作系统 SOSC 的效率，人们提出了批处理操作系统。在仔细考察了 SOSC 后，人们发现，SOSC 效率之所以低下，是因为计算机总是在等待人的下一步动作，而人的动作总是很慢。因此，人们觉得，如果将人的因素拿走，让所有的人先想好自己要运行的命令，列成一个清单，打印在纸带上，然后交给一个工作人员来一批一批地处理，效率不就提高了吗？这样就形成了我们所说的批处理操作系统。

批处理操作系统针对的是第二代通用计算机，如 IBM 的 1401 和 7094 等，通过去除人机交互达到 CPU 和输入输出利用率的改善。批处理的过程是：用户将自己的程序编在卡片或纸带上，交给计算机管理员。管理员在收到一定数量的用户程序后，将卡片及纸带上的程序和数据通过 IBM1401 机器读入，并写到磁带上。这样每盘磁带通常会含有多个用户的程序。然后，计算机操作员将这盘磁带加载到 IBM7094 上，一个一个地运行用户的程序，运行的结果写在另一个磁盘上。所有用户程序运行结束后，将存有结果的磁盘取下来，连到 IBM1401 机器上进行结果打印，然后就可以将打印结果交给各个用户了。图 2-1 描述了批处理的过程。

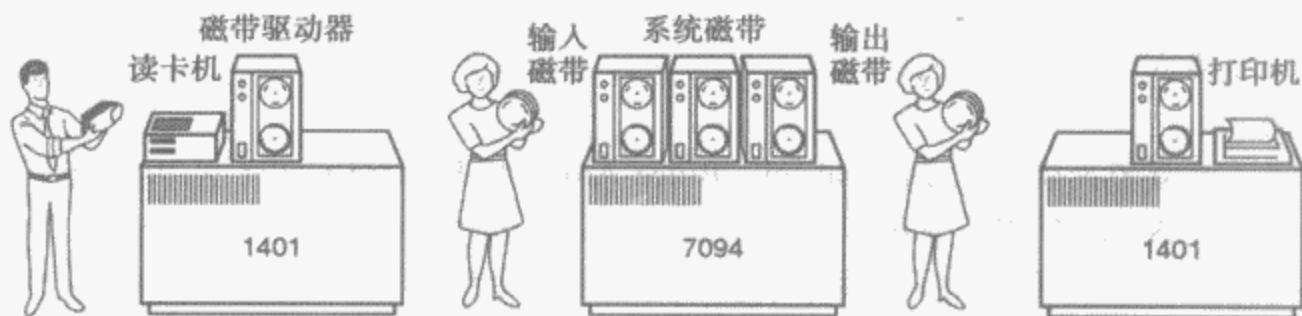


图 2-1 批处理系统示意图（来源：参考文献 [3]）

很显然，在批处理下，操作系统的功能和复杂性均得到提升。在 SOSC 环境下，每个用户自己控制程序的开始和结束。而在批处理下，很多用户的程序一个接一个地存放在磁带上，用户本人并不在场，无法自己控制程序的开始和结束。而这个任务就交给了批处理操作系统。负责这个任务的操作系统功能就称为批处理监视器（batch monitor）。整个批处理操作系统就是由批处理监视器和原来的操作系统库函数组成（见图 2-2）。

批处理监视器的功能就是按部就班地执行用户的程序。这个时代的操作系统仍然只能在同一时间执行一个程序，但此时文件的概念已经出现。之前在 SOSC 诞生时期没有文件的概念。为什么到批处理时期出现了文件

的概念呢？因为磁带上的多个用户程序必须以某种方式进行隔离，这需要一个抽象的东西来区分一下。这个抽象的东西不是别的，就是文件。除了文件管理外，此时的操作系统还能够管理读卡机，磁带，打印机等。此种操作系统的任务就是加载一个程序、运行、打印结果，然后执行下一个程序。批处理操作系统的两个部分的关系也很清楚：一部分是控制程序执行，一部分是支持程序执行。

批处理操作系统的重要实例有 IBM 开发的 FORTRAN 监视系统 FMS，用于 IBM 709；IBM 开发的基于磁带的工作监控系统 IBSYS，用于 IBM 7090 和 7094；密歇根大学开发的 UMES（密歇根大学执行体系统），用于 IBM7094。

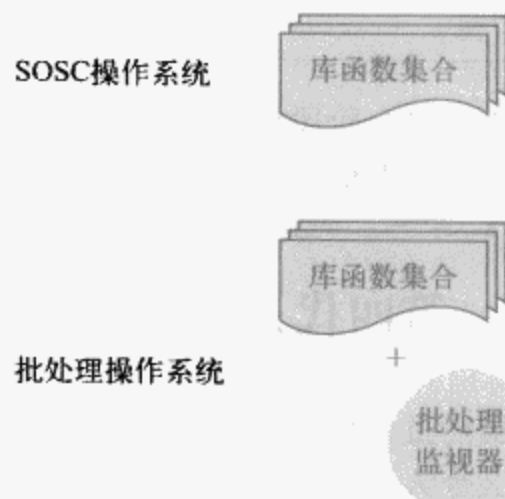


图 2-2 SOSC 和批处理操作系统之比较

密歇根大学执行系统 UMES：操作系统的黎明

这个时候，世界上最先进的计算机是 IBM7094。作为礼物，IBM 分别给密歇根大学（UM）和麻省理工学院（MIT）各捐赠了一台。密歇根大学靠湖：密歇根湖和伊犁湖，麻省理工学院靠海：大西洋。IBM 的高管喜欢搞帆船比赛。而每次搞帆船比赛，需要使用计算机来安排赛程、计算成绩、打印名次等。因此，IBM 在捐赠机器给密歇根和 MIT 时有一个要求：平时归学校用（MIT 的机器还需要与新英格兰周围的学校如达特茅斯学院等共享），一旦来帆船比赛就得停下一切计算任务为 IBM 服务。这当然使得学校很恼火。因为

那个时候很难在程序执行中间停下来，将中间结果保留等以后再执行。只要停下来，就要从头来过。

于是，密歇根大学的 R. M. Graham, Bruce Arden 和 Bernard Galler 在 1959 年开发出了当时很有名的 MAD/UMES 系统，即密歇根算法译码器和密歇根大学执行系统。密歇根算法译码器是一种可扩展的程序设计语言，而密歇根大学执行系统是一个能够保存中间结果的操作系统。有了这个系统，密歇根的计算机运行基本就不受 IBM 帆船比赛所造成的中断的影响。MIT 在知道这个系统后，将其安装到了自己的 7094 机器上，而 MAD 编程语言随后又进一步移植到 Philco、Univac 和 CDC 等机器上，其很多功能后来被加入到 FORTRAN 语言里。

驱动这个阶段操作系统发展的动力是改善效率。因为机器的昂贵，我们不能容忍机器在操作员思考或 I/O 设备工作期间闲置起来。

2.4 第四代：多道批处理操作系统（20 世纪 60 年代）

虽然批处理操作系统通过无需人机交互过程而在一定程度上提高了计算机的效率，但还是不那么令人满意。因为，CPU 和输入输出设备的运行是串行的，即在程序进行输入输出时，CPU 只能等待。CPU 需要不断地探询 I/O 是否完成，因而不能执行别的程序。再看一遍图 2-1：磁带上的程序需要先读进来，程序才能执行，执行完了又需要写到另一个磁带上。读写磁带的时候 CPU 是不工作的，这就是很大的浪费。

这个时候，由于 I/O 设备的运行速度相对于 CPU 来说实在太慢，这种让高速设备等待低速设备的状况令人颇感痛心。人们又想，能否将 CPU 和 I/O 进行并发呢？即在一个程序输入输出时，让另一个程序继续执行。换句话说，能否将 CPU 运行和输入输出设备的运行重叠起来而改善整个系统的效率呢？答案是肯定的，不过需要付出代价。因为 CPU 和 I/O 重叠需要我们将多个程序同时加载到计算机内存里，从而出现了所谓的多道批处理操作系统。

在多道批处理操作系统时代，同一时间可以运行多个程序（宏观上），但控制计算机的人还是只有一个，即用户将自己的程序交给计算机管理员，由管理员负责将用户的程序加载到计算机里并执行。由于多个程序同时执行，操作系统需要能够在多个程序（工作）之间进行切换，并且能够管理多个输入输出设备，同时还需要能够保护一个进程不受另一个进程干扰。

显而易见，操作系统的功能和复杂性都比简单批处理时要复杂得多：既要管理工作，又要管理内存，还要管理 CPU 调度。

OS/360：划时代的多道批处理操作系统

典型的多道批处理操作系统是 IBM 的 OS/360，它运行在 IBM 的第三代计算机 System/360、System/370、System/4300 等之上。OS/360 在技术上和理念上都是划时代的操作系统，

但在商业上没有使用。因为它有很多错误，可这是避免不了的。划时代的，崭新的东西，你很难一次做到完美。内存的分段管理也是在这时候引进的。OS/360 虽然有很多错误，但是不妨碍其作为一个划时代的操作系统，而且同时支持商业和科学应用，之前只作科学计算。IBM 随后对 OS/360 进行了完善，逐渐演变为一个功能强大、性能可靠的操作系统。该操作系统提供了资源管理和共享，允许多个 I/O 同时运行，CPU 和磁盘操作可以并发。

驱动这个阶段操作系统发展的动力仍然是改善效率。因为机器的昂贵，我们不能容忍机器 (CPU) 在 I/O 设备工作期间闲置下来。同时，我们对计算机的要求也开始多起来。因此，这个阶段还伴随着对用户不断增长的要求进行满足。

2.5 第五代之一：分时操作系统 (20 世纪 70 年代)

多道批处理操作系统的出现使计算机的效率 (主要是吞吐率) 大大提高。不过这时人们又提出了另外一个问题：将程序制作在卡片上交给计算机管理员统一运行，将使得用户无法即时获知程序运行的结果。而这是一个大问题。想想如果你编了一个程序，却需要让别人去运行，并等上若干天才能知道结果，这个滋味显然不好受。万一计算机管理员疏忽了，忘记运行你的程序，或者操作错误，导致程序丢失，情况就更加糟糕。

基于上述考虑，人们就想，能否让人回到计算机前面来，每个人自己管自己的程序，但是，大家的程序可以同时运转。人的因素又引了回来。这看上去与原始的 SOSC 似乎一样，但有个关键的不同：多个人同时连在计算机上，每个人看作是另外的一个 I/O 终端而已。每个用户拥有一个终端显示器，这些终端显示器经过 RS232 穿行线缆与计算机连接。终端显示器只能接收和有限的发送文本命令和信息。计算机在所有连接的终端用户之间进行分时，即分给每个人有限的时间，只要时间到了，就换一个进程。这种时分切换下的操作系统就是分时操作系统。

在分时操作系统下，任意时间可以运行多个程序，且用户直接与计算机交互，当场调试程序。这就和单一操作员单一终端不一样了。就从人本转成物本。单一操作员单一终端的情况下，一切等着人。以前执行一条命令就等人，分时系统是人等机器。这个模型带来一个直接的结果，就是机器再也不用等你，等你想问题时机器就切换到别的程序，等你想完了机器再切换回来，接受你的输入。就这样，计算机就在很多人之间来回转，你敲个命令就响应，然后切换走。如果时间掌握的好，用户输入完一个命令计算机正好回来，用户就没有等待的时间开销。当然，如果一个用户打字足够快，可能会觉得计算机慢；如果打字足够慢，就有可能觉得计算机很快。不同的人感觉有可能完全不同。

显然，和前面几代的操作系统相比，分时操作系统要复杂得多。相比于多道批处理系统，最主要的变化是资源的公平管理。在多道批处理下，公平不公平没有人知道。大家交了工作后只管回家等结果。至于自己的程序排在谁前面谁后面，或者占用了多少 CPU 时间是无关紧要的。现在，大家都坐在计算机显示终端前面，任何的不公平将立即感觉到。因此，公平的管理用户的 CPU 时间就变得非常重要。除此之外，池化 (pooling)、互斥、进程通信等机制相继出

现,使得分时操作系统的复杂性大为增加。

傲慢的代价: MULTICS 操作系统

分时操作系统里面最为有名的应该是 MULTICS 和 UNIX。前面讲过,IBM 在其捐赠 7094 给 UM 和 MIT 时附加了一些条件,而这些附加条件弄得学校非常恼火,但又不能拒绝(想想别的学校吧,他们连这种恼火的机会都没有)。为此,密歇根开发了 UMES 系统以应对这个问题。MIT 则将 UMES 移植到了自己的 7094。后来觉得光能够保存中间结果还不是最好的办法,毕竟频繁地保存中间结果等帆船比赛结束后再进行重载仍是麻烦,于是就想来开发一个可支持多用户的分时操作系统,以便一劳永逸地解决这个问题。这个时候 MIT 想到了密歇根的 R. M. Graham (见图 2-3),将其请来。在 R. M. Graham 的主持下,来自于贝尔实验室、DEC (美国的数字仪器公司, Digital Equipment Corporation) 和 MIT 的设计人员同仇敌忾, (针对 IBM 的傲慢), 一起努力,开始了 MULTICS 分时操作系统的研制。

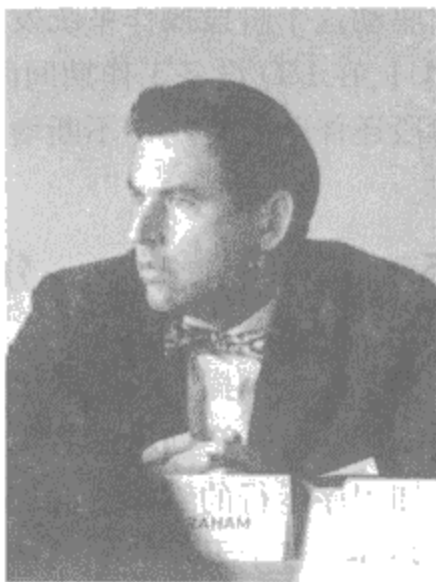


图 2-3 R. M. Graham

不过,在 MULTICS 还没有开发出来的时候,开发团队内部出现了分歧,贝尔实验室的几个人越来越看不惯 DEC 和 MIT 的人,觉得这伙人的思想方法跟自己不一样,做的东西不专业,觉得和他们一起做研究把自己的水平降低了,就自立门户,搞出了 UNIX,并因此获得了图灵奖。而 UNIX 的出现,使得 MULTICS 从一面世,就不能挺立,真是中国历史上“既生瑜、何生亮”在计算机操作系统历史上的完美演绎。不过,MIT 最后还是搞出了一个应用于部分商用领域的分时操作系统 CTSS (compatible Time sharing system)。

分时操作系统通常运行在第三代机 PDP, VAX 和 CRAY 上,其中 PDP, VAX 是 DEC 公司生产的,不过 DEC 已经不复存在。CRAY 是 CRAY 公司生产的。

驱动这个阶段操作系统发展的动力是响应时间和对越来越多资源的管理。因为机器昂贵,我们不能容忍机器 (CPU) 在 I/O 设备工作期间闲置下来。同时,因为人的时间宝贵,我们不能容忍人们坐在机器前进行漫长的等待。因此,我们发明了分时操作系统来解决这两个问题。因为分时而引入的多道程序设计,又造成操作系统的空前复杂,我们需要应对竞争、通信、死锁、保护等一系列的新功能。因此,操作系统在本阶段变得相当复杂。

2.6 第五代之二: 实时操作系统

随着人类技术的进步,计算机得到了广泛的应用。其中的一种应用称为进程控制系统,即使用计算机对某些工业进程进行监视,并在需要的时候采取行动。所有这些系统都具备一个特

点：计算机对这些应用必须在规定时间内作出响应，不然就有可能发生事故或灾难。例如，在工业装配线上，当一个部件从流水线上一个工作站流到下一个工作站时，这个工作站上的操作必须在规定时间内完成，否则就有可能造成流水线瘫痪，影响企业的生产和利润。又例如，在导弹防卫系统中，对来袭导弹的轨迹计算必须在规定时间内完成，否则就可能被来袭导弹击中而无法作出反应。其他对计算机响应时间有要求的系统包括核反应堆状态监视系统、化学反应堆监视系统、航空飞行控制系统等。

这种对计算机响应时间有要求的系统通常称为临界系统或应用。为了满足这些应用对响应时间的要求，人们就开发出了实时操作系统。实时操作系统是指所有任务都在规定时间内完成的操作系统，即必须满足时序可预测性（timing predictability）。这里需要提请读者注意的是，实时系统并不是反应很快的系统，而是反应具有时序可预测性的系统。当然了，在实际中，实时系统通常是反应很快的系统，但这是实时系统的一个结果，而不是其定义。

显然，实时操作系统的最重要部分就是进程或工作调度。只有精确、合理和及时的进程调度才能保证响应时间。当然，对资源的管理也非常重要。没有精密复杂的资源管理，确保进程按时完成就成为一句空话。另外，基于其使用环境，实时操作系统对可靠性和可用性要求也非常高。如果在这些方面出了问题，时序可预测性将无法达到。

实时系统通常又分为软实时系统和硬实时系统。软实时系统在规定时间得不到响应所产生的后果是可以承受的，如流水装配线。即使装配线瘫痪，不就是损失点钱吗？而硬实时系统在得不到实时响应可能产生不能承受的灾难，如导弹防卫系统。如果反应迟钝，结果就可能是人命损失。

商业实时操作系统的代表有 VxWorks 和 EMC 的 DART 系统。

2.7 第六代：现代操作系统（1980 年以后）

在 80 年代后期，计算机工业获得了井喷式的发展。各种新计算机和新操作系统不断出现和发展，计算机和操作系统领域均进入到了一个百花齐放、百家争鸣的时代。尤其重要的是工作站和个人机的出现，使计算机大为普及。独享计算机也可以负担得起。这个时候的操作系统代表有：DOS、Windows、UNIX、Linux 和各种主机操作系统，如 VM、MVS、VMS 等。DOS、Windows、UNIX、Linux 通常称为开放式系统操作系统，分别运行在 PC、VAX 和工作站上。操作系统也重新回到子函数库的状态。

随着硬件越来越便宜，个人机出现在人们的视野中。人们可以拥有自己的电脑，无需与别的人分享。在刚刚出现个人机的时候，拥有个人机的人感觉很好，而那些需要与别人共享小型机的人则感觉不好。由于个人机由用户一个人独享，分时操作系统的许多功能就无需存在。因此，个人机操作系统又变回到了标准函数库系统。这个时候最有名的当然是 DOS、Windows 3X、苹果机操作系统（MacOS）等。

但在独享了一阵个人机后，人们发现，没有分时功能的操作系统使一些事情做不了。因为，虽然只有一个人在用机器，但这个人可能想同时做好几件事，例如，同时运行好几个程序，没有分时功能这是不可能的。于是，人们觉得需要对个人机操作系统进行改善，将各种分

时的功能又加了进去。

这时候就需要对程序进行保护，因为现在运行的多个程序，虽然都是你的东西，但是也不能混淆。于是，Windows NT 出现了，Xenix 出现了，Ultrix 出现了。

这个时候的另外一个特征是网络的出现。网络触发了网络操作系统和分布式操作系统的出现。对于网络操作系统来说，其任务是将多个计算机虚拟成一个计算机。传统的网络操作系统是在现有操作系统的基础上增加网络功能，而分布式操作系统则是从一开始就把对多计算机的支持考虑进来。

后来分布式操作系统出现了，因为网络出现了。我们现在虽然有很多电脑，但是很多处于闲置，这就是一个极大的浪费，我们希望空闲的都能利用起来。分布式计算出现的原因是希望把任务分开，得到的结果是计算资源的集合，这让很多计算机看上去像一个。有两种分布式系统，一个是网络操作系统，就是在传统操作系统上加一个补丁，一个是分布式操作系统。网络操作系统就是打补丁，分布式操作系统是重新设计的一套，所以比网络操作系统效率高。

2.8 操作系统的演变过程

计算机操作系统的演变可以从三条发展线索来看。这三条发展线索分别是主机操作系统、服务器操作系统和个人机操作系统。图 2-4 简略地画出了这三条线索上操作系统的演变历史。

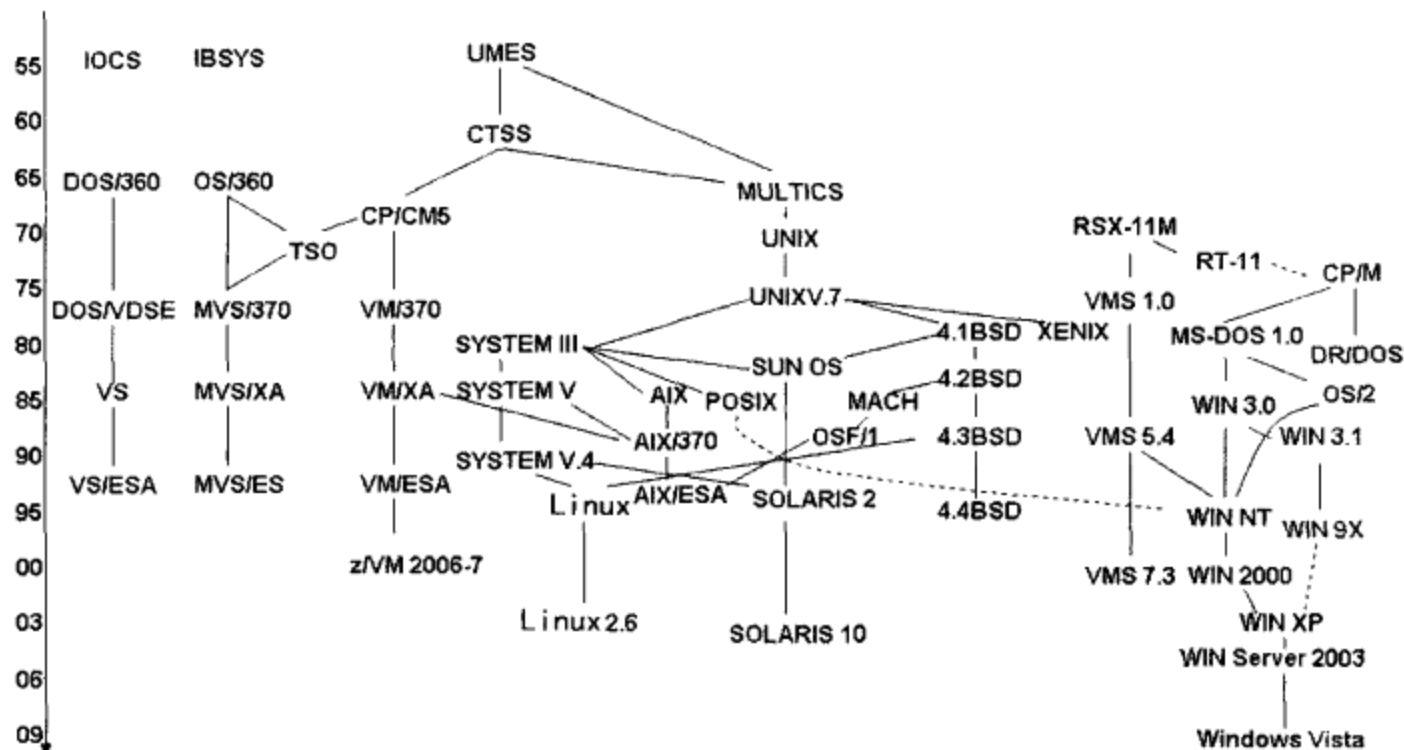


图 2-4 操作系统的历史演变

图 2-4 最左面的三列代表主机操作系统的演变，最右面的两列代表个人机操作系统的演变，中间的列为服务器操作系统的演变。当然，到最后，这三大块的界限也不是很清楚。

主机操作系统的演变从输入输出控制系统 IOCS 和 IBSYS 开始，经历 OS/360 的里程碑式的突破，逐步演变为 VS、MVS 和 VM 三个系列（其中 VM 系列还吸取了 UMES 和 CTSS 的某

些特征)。目前 IBM 是这三个系列的开发商和运营商,其 VM 操作系统经历多代后已经变得十分可靠。美国大型金融证券公司都在使用它们。

服务器操作系统的演变从 UMES 开始,经 CTSS 演变为 IBM 的 MVS 和 VM 操作系统,经 MULTICS 演变为 UNIX 系统。在 80 年代初,UNIX 一分为二:由 AT&T (美国电话电报公司)提供的 System 系列和由 UCB 提供的 BSD 系列。XENIX 是微软公司为 PC 机而移植的 AT&T 版的 UNIX 操作系统。AT&T 是 UNIX 的鼻祖,UCB 则在美国国防部的支持下开发了 BSD (伯克利软件分配)系列。

IBM 和斯坦福大学看到 AT&T 和 UCB 的 UNIX 软件后,也不甘示弱,分别研发了 AIX 和 SUN OS (SUN 是斯坦福大学网络 Stanford University Networks 的缩写)。这样 UNIX 就形成了 4 个系列:美国电话电报公司的 System 系列、国际商用机器公司的 AIX 系列、斯坦福大学网络的 SUN OS 系列和伯克利加州大学的软件分配 BSD 系列。

90 年代中期,在美国国防部停止了对 BSD 的支持后,UCB 停止了 BSD 系列,而 AT&T 也在与 BSD 焦头烂额的较量中放弃了 System 系列。LINUX 则趁着 AT&T 和 UCB 忙于与对方较量的时候发展出来。卡内基梅隆大学 (Carnegie Mellon University) 在看到 UCB 和斯坦福都研发了 UNIX,慌忙做了个 MACH 操作系统。MACH 为微内核操作系统,在学术界得到了一定的使用,但由于其运行效率低下而没有获得商业上的广泛应用。这样 UNIX 的商业使用版本就剩下 AIX、SOLARIS 系列 (SUN OS 的后续) 和 LINUX 系列。后来由于惠普公司 (HP) 加入到服务器行列使得 UNIX 家族又增加了一个版本:HP-UX。在 UNIX 家族里面,SUN 公司濒临灭亡的命运令 SOLARIS 的前途堪忧。

个人操作系统的演变可以说是从 DOS 开始的。微软在 1980 年以 100 美元的成本买断了 DOS 的版权。而 DOS 的功能很简单:文件没有文件夹,所有文件都在一个地方,谁都可以删除操作系统。当然那时候也没有那么多人从事破坏活动。在看到苹果的 MacOS 的图形界面后,微软给 DOS 增加了一个图形界面,称之为 WINDOWS。WINDOWS 在发展了几个版本后,到 WINDOWS98 时,微软改变了战略。因为到目前为止,所有的 WINDOWS 并不是真正的操作系统,而是覆盖在 DOS 上的一个用户图形界面,不能支持多道编程。微软高管比尔盖茨亲自打电话给 DEC 的 David Cutler,请其过来主持新一代 WINDOWS 操作系统研发。

David Cutler 是 DEC 公司 VMS 操作系统的主要设计人员。他从 DEC 带过来一批人到微软工作,设计出了 WINDOWS NT 操作系统。这是一个真正支持多道编程的操作系统。WINDOWS NT 继承了 VMS 的优良结构和 WINDOWS 3X 的图形界面,一推出就获得了市场的接受。WINDOWS NT 经过几代演变,成为现在的 WINDOWS Vista。WINDOWS 操作系统系列也从单一的支持个人机演变为支持个人机和服务器的“双料”操作系统。

操作系统的分类

操作系统基本上可以分为主机操作系统,如 OS/260, OS/390, CTSS; 服务器操作系统,如 UNIX、Windows 2000、Linux; 多 CPU 计算机操作系统,如 Novell Netware; 个人计

计算机操作系统 Windows 2000、Windows XP、MacOS；实时操作系统，如 VxWorks、DART；嵌入式操作系统，如 Palm OS、Windows CE、TOPPER 等。

同一台计算机上可以运行不同的操作系统，而同一个操作系统也可以运行在不同的计算机上。例如，个人机上可以运行的操作系统有 DOS、Linux、NeXTSTEP、Windows NT、SCO UNIX 等；DEC VAX 计算机上可以运行的操作系统有 VMS、Ultrix-32、BSD UNIX 等。UNIX 操作系统可以在 XENIX 286、APPLE A/UX、CRAY-Y/MP、IBM 360/370 等计算机上运行；Windows NT/XP 可以在英特尔 386 和 Itanium、DEC 的 Alpha、摩托罗拉的 PowerPC 和 MIPS 计算机的 MIPS 上运行。

当然了，运行在不同机器上的 UNIX 版本并不一样，例如运行在 IBM 360/370 上的 UNIX 是 Amdahl UNIX UTS/580 和 AIX/ESA。运行在 CRAY-Y/MP 计算机上的 UNIX 是 AT&T System V。表 2-1 列出的是互联网上用户对各个操作系统的评价。从表中看出，使用最为普遍（投票用户数超过 10000）的操作系统是 Linux、MacOS X、OpenBSD、SOLARIS、UNIX、VMS 和 Windows，其中用户评价最高的操作系统是 VMS，其反对票数与赞成票数的比率几乎是 0，用户评价最低的是 OpenBSD，反对票数比赞成票数的 25 倍还多。

表 2-1 截止到 2009 年 2 月 16 日 全球互联网用户操作系统评价

操作系统	反对票数	赞成票数	反对赞成比率
AmigaOS	63	75	0.84
BeOS	135	283	0.48
FreeBSD	673	9090	0.07
Linux	213000	379000	0.56
MVS	63	168	0.38
MacOS	1748	1517	1.15
MacOS X	35200	29900	1.18
NetBSD	257	654	0.39
Netware	66	135	0.49
OS/2	58	211	0.27
OS/400	10	16	0.63
OpenBSD	19800	781	25.35
SOLARIS	10600	1841	5.76
UNIX	14800	24200	0.61
VMS	103	48840	0.002
Windows	433000	71500	6.05

从互联网用户对各种商用操作系统的评价来看，每种操作系统都有人不满意。这是因为操作系统是一个非常复杂的软件，其涉及的技术繁多，设计出好的操作系统十分不容易。

2.9 操作系统的未来发展趋势

随着计算机的不断普及,操作系统的功能会变得越来越复杂。在这种趋势下,操作系统的发展面临两个方向的选择:一是向微内核方向发展,二是向大而全的全方位方向发展。微内核操作系统虽然有不少人在研究,但在工业界获得认可的并不多。这方面的代表有 MACH 系统。对工业界来说,操作系统是向着多功能、全方位方向发展。Windows XP 操作系统现在有 4000 万行代码,某些 Linux 版本有 2 亿行代码, Solaris 的代码行数也不断增多。鉴于大而全的操作系统管理起来比较复杂,现代操作系统采取的都是模块化的方式,即一个小的内核加上模块化的外围管理功能。

例如,最新的 Solaris 将操作系统划分为核心内核和可装入模块两个部分。其中核心内核又分为:系统调用、调度、内存管理、进程管理、VFS 框架、内核锁定、时钟和计时器、中断管理、引导和启动、陷阱管理、CPU 管理;可装入模块又分为:调度类、文件系统、可加载系统调用、可执行文件格式、流模块、设备和总线驱动程序等。

最新的 Windows 将操作系统划分成内核(kernel)、执行体(executive)、视窗和图形驱动、可装入模块。Windows 执行体又划分为: I/O 管理、文件系统缓存、对象管理、热插拔管理器、能源管理器、安全监视器、虚拟内存、进程与线程、配置管理器、本地过程调用等。而且, Windows 还在用户层设置了数十个功能模块,可谓功能繁多,结构复杂(见图 2-5)。

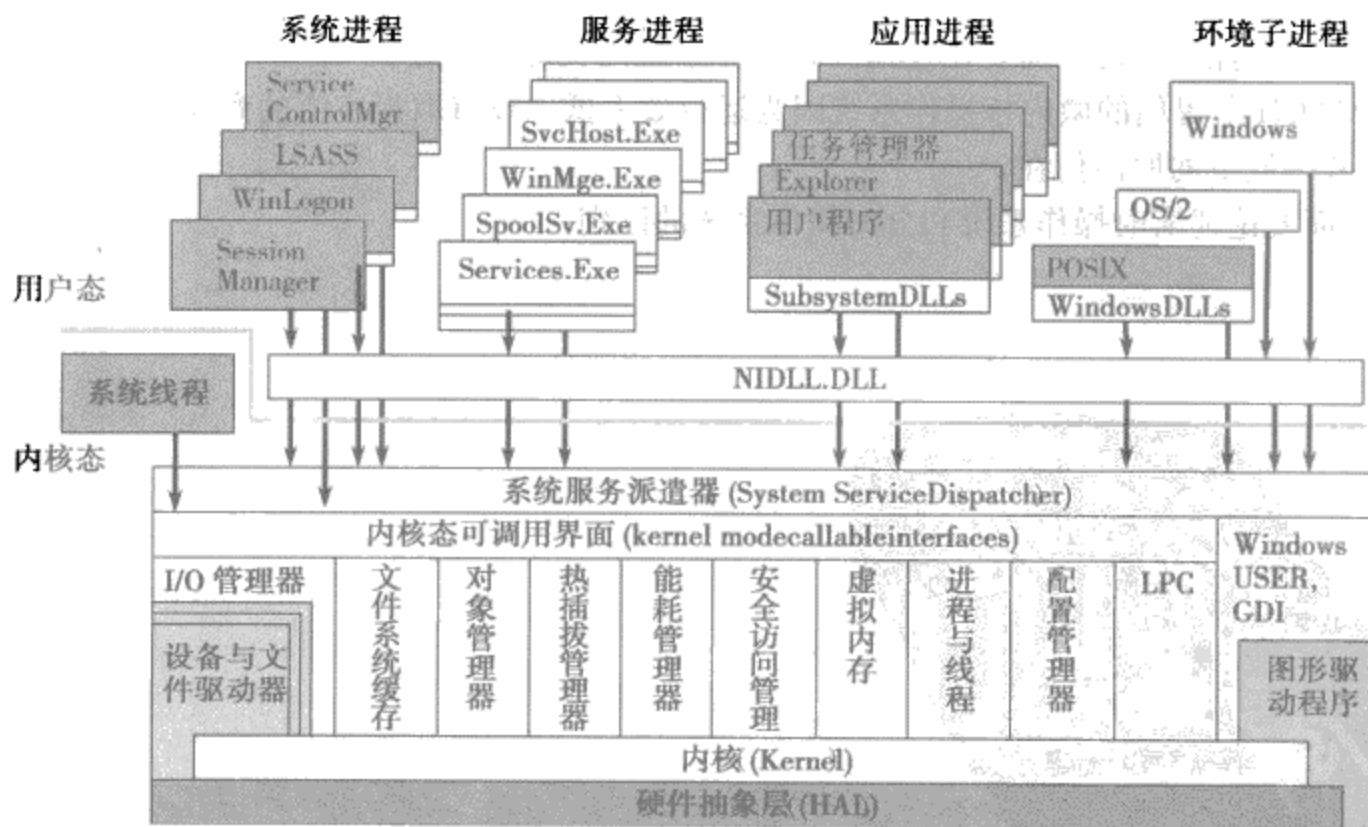


图 2-5 Windows 2000/XP/2003 系统结构(来源:参考文献[8])

随着人们对信息安全重视程度的不断提升,如何构建可靠、可用和安全的操作系统将成为一个十分重要的课题。而对可靠、可用和安全的追求无疑将使操作系统更为复杂,操作系统的

规模也将不断增大。从 UNIX 的 1400 行代码到 Windows XP 的 4000 万行代码，这完全是一种爆炸性增长。而爆炸性增长的后果就是，没有什么人能够完全理解一个完整的操作系统，而这种状况又将限制操作系统的可靠、可用和安全性。当然了，人们可以采用各种软件工程的方法和手段来改善这种状况。但无论如何，持续的爆炸性增长恐怕是难以为继的。

综上所述，不要预测将来。那就让我们所有关心操作系统的人拭目以待吧。

思考题

1. 请列出你曾经用过的所有操作系统。哪个操作系统你觉得最好？为什么？
2. 你用过的操作系统里面，你感觉哪个操作系统最好？解释你的答案。
3. 计算机从过去单一控制终端单一操作员到现在的个人机，似乎我们转了一个圈。是不是我们人总是喜欢反复无常呢？请阐述你对这种否定之否定的观点。
4. 虽然我们不赞成对未来进行预测，但你是否对操作系统的未来演变有自己的看法呢？
5. MULTICS 的出现在很大程度上是由于 IBM 的傲慢，你认为人的傲慢在操作系统发展过程中占有什么样的角色呢？
6. 驱动操作系统发展的主要动力有哪些？它们是如何驱动的？
7. 很多人都说，没有操作系统的计算机是一堆废铁，无法运转。但在计算机刚诞生的时候，谁也不知道操作系统这回事。那个时候的计算机为什么在没有操作系统的情况下能够运转呢？它们又是如何运转呢？
8. 操作系统根据其运行的计算机硬件结构不同而分为主机操作系统，服务器操作系统和个人机操作系统。简要论述这三种操作系统的关键不同点。
9. MACH 所提倡的微内核操作系统因为运行效率低下没有取得广泛的商业应用，你认为其效率低下的原因何在？
10. 简要论述实时操作系统和分时操作系统的区别。

第3章 操作系统基本概念

引子：“差不多”精神

有一个小幽默也许读者听过，说是三个数学家和三个软件专家在一列火车上相遇。攀谈中，六个人发现大家都是参加同一个会议。然而让软件专家们吃惊的是，三个数学家只买了一张车票，而他们自己却各买了一张票。

三个软件专家于是问数学家：“你们三个人只买一张票，等会列车员查票你们怎么应付呢？”（这里需要指出的是，西方的火车站不查票，而是在车上查票。）

三个数学家回答说“这你们就不懂了吧。你们看好了，学习学习。”

过了不久，果然开始查票。只见三个数学家急急忙忙跑到厕所里面，将门从里面反锁上。查票员过来后敲打厕所的门，问里面有人吗。里面传来一个声音“有人，但正在上厕所，无法开门，能否将车票从门底下递出来给查票员检查？”这个时候厕所门下面递出来一张票。查票员看了没有任何问题，于是就离开了。

三个软件专家看完了这一幕，十分惊叹。心想，数学家就是比研究软件的人聪明，我们怎么从来没有想到这一招呢？于是，他们相约回程再同乘一次车。

过了几天，开完会后，六个人在回程途中又聚到一起。见面后，三个软件专家就迫不及待地告诉数学家们：“我们这次只买了一张票。你们是否也是只买了一张票？”

只见数学家们微笑着，不动声色地回答说：“我们这次没有买票。”

软件专家一听，不敢相信自己的耳朵。接着问道：“那你们这次怎么应付查票呢？”

数学家们丢下一句话“你们自己想吧。”

软件专家想了半天，实在是想不出来有什么办法可以应付查票员。没有办法，只好硬着头皮来问数学家：“我们想不出来，你们告诉我们吧。”

数学家们仍然微笑着：“你们的脑袋不够用了吧。看我们再教你们一招。”

过了不久，查票开始了。三个软件专家急急忙忙跑到厕所里面，将门从里面反锁上。这时有人敲厕所的门，说查票了，里面有人没有。里面传来一个声音“有人，但

正在上厕所，无法开门，能否将车票从门底下递出来检查？”这个时候厕所门下面递出来一张票。

门外的人接了车票，却再也没有递回来……

这个幽默对于学软件的人来说，也许不会感到很幽默。可对于学数学的人来说，听了后感觉很好。做软件的人认为是数学家挣钱挣不过他们，于是编造出这么个故事来，乘机阿 Q 一把。

不过沉下心来想一想，发现数学家的做法还真有点道理。这个道理不是别的，而是因为数学是严谨的学科，一切都以精确为追求（这里指纯数学，不包括那些不被认为是数学的应用数学）。而软件呢，却没有任何精确可言，是十足的“模糊”学科。因为，软件是一门人造学科，它没有对与错（这里指的是同一功能的不同实现，而不是说程序不可能出现错误），只有好与坏。我们设计软件的时候，也是觉得差不多就可以了，而没有什么精确的追求。

如果不信，就看看我们是如何分析算法的吧。我们使用所谓的渐近分析，将系数和非决定项都抛去了。所谓的“只要数量级对就差不多”。至于软件可靠性、健壮性和成本估算，那就更不用说了，连差不多都不如，而是差很多。

操作系统作为一种软件，自然也是差不多就可以了，其中的许多设计都是各种折中的结果，到处体现着差不多精神。读者在后面学习操作系统时只要留心观察，就会发现这些差不多的存在。而要理解这种“差不多”系统，自然需要抱着一种“差不多”的态度。如果一切吹毛求疵，学习操作系统会非常痛苦，也很难精确（不好意思，用了一次“精确”）把握操作系统的精髓。

令人欣慰的是，中国人自古就有“差不多”的思维逻辑，因此采纳“差不多”的态度对于中国人是很容易的事情。从这种角度看，学好操作系统是水到渠成的事情。（这里需要强调的是，作者并不是说计算机学科里面没有任何需要精确的时候，而是说在计算机学科里面体现“差不多”的地方非常多，以至于整个学科都带有某种“差不多”的精神）。

我们前面说过，操作系统是一个软件，它运行在硬件上，又为更加上层的应用软件提供服务。因此，对底层硬件的了解将帮助我们更好地把握操作系统。下面我们就从计算机硬件开始介绍，探讨一下操作系统的主要概念。

3.1 计算机硬件基本知识

从概念上讲，计算机的结构非常简单：首先布置一根总线，然后将各种硬件设备挂在总线上。所有这些设备都有一个控制设备，外部设备都由这些控制器与 CPU 通信。所有设备之间的通信均需通过总线，如图 3-1 所示。图中的粗线条为总线。

为了提高计算机的效率，人们又设计出了流水线结构，即仿照工业流水装配线，将计算机的功能部件分为多个梯级，并将计算机的每条指令分拆为同样多个步骤，使每条指令在流水线上流动，到流水线最后一个梯级时指令执行完毕。流水线上的每个梯级都可以容纳一条指令同时执行，如图 3-2 所示。

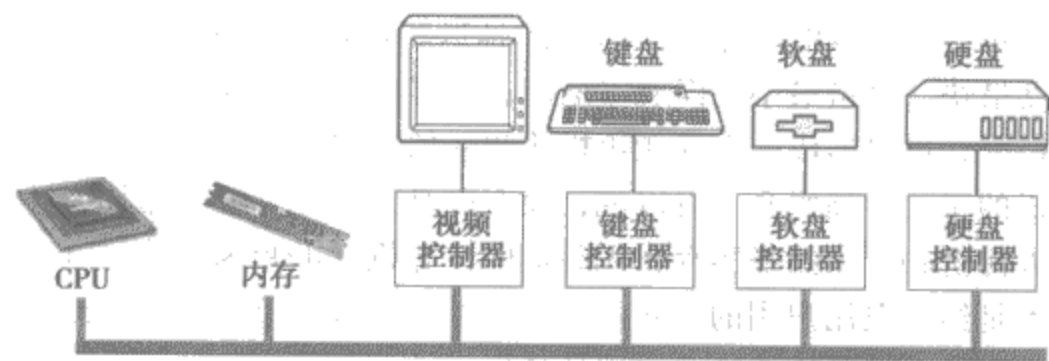


图 3-1 计算机结构概览图

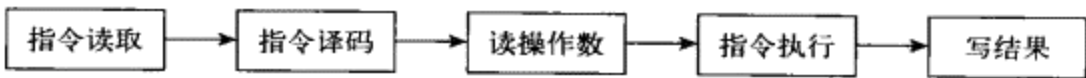


图 3-2 一个 5 个梯级的流水线结构

为了进一步提高计算机的效率，在流水线的基础上，人们又发明了多流水线、超标量计算和超长指令字等多指令发射机制。这些机制的发明在提升计算机效率（主要是吞吐量）的同时，也极大地增加了计算机结构的复杂性，并对操作系统和编译器都提出了更高的要求。

图 3-3 描述的是一个超标量发射的体系结构。这个结构有两队指令读取和译码单元，三个执行单元。通过一个指令保持缓冲区，就可以实现多路复用（multiplex）和反多路复用（demultiplex），从而提高系统每个功能单元的利用率和整个系统的吞吐量。

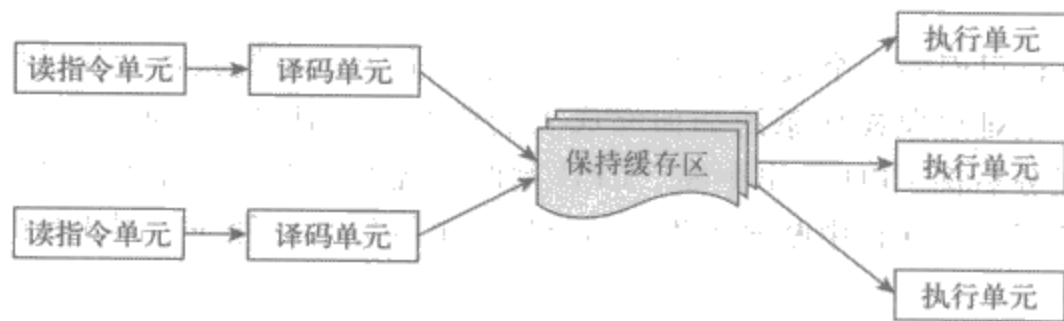


图 3-3 一个 4 个梯级的超标量发射结构

除了指令执行单元外，计算机里面的另一个重要部件是指令的存放单元，称为存储架构。存储架构包括了缓存、主存、磁盘、磁带。有的情况下还存在多级缓存和外部光盘。图 3-4 描述的是一个包括寄存器的 5 级存储介质构成的存储架构。

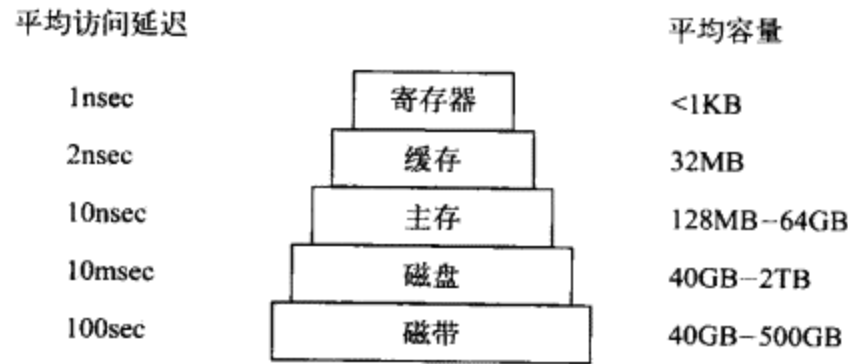


图 3-4 典型的 5 级存储结构

从寄存器到磁带，每一级的存储媒介的访问延迟和容量均依次增大，而价格却依次降低。寄存器的访问速度最快，容量最小，但成本最高；磁带的访问速度最慢，容量最大，成本却最低。通过合理的搭配，可以形成一个性能价格比颇佳的存储架构。

磁盘是计算机的主要存储媒介。可以说，没有磁盘，计算机就不称其为计算机，或者说计算机的用处就要大打折扣。虽然确实存在无磁盘的计算机（diskless 计算机），但这些计算机都有特别之用，并不是给一般用户用的。磁盘从概念上看非常简单，每个磁盘有多块盘片，盘片两面都可以存储。图 3-5 描述的是典型的磁盘结构。

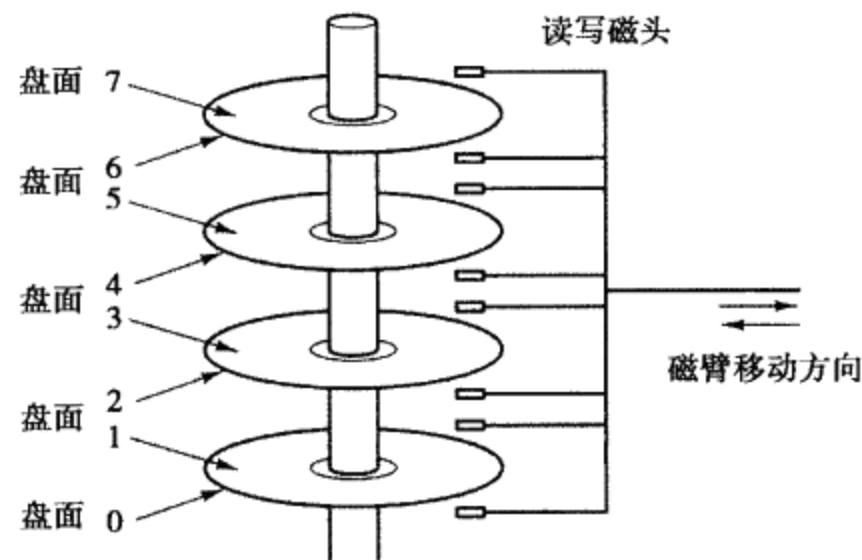


图 3-5 典型的磁盘内部结构

中断是计算机里面的一个最为重要的机制，它也是操作系统获得计算机控制权的根本保证。没有中断，很难想象操作系统如何完成人们所赋给的任务。中断的基本原理是：设备在完成自己的任务后向 CPU 发出中断，CPU 判断优先级，然后确定是否响应。如果响应，则执行中断服务程序，并在中断服务程序执行完后继续原来的程序。图 3-6 简单地描述了中断机制。

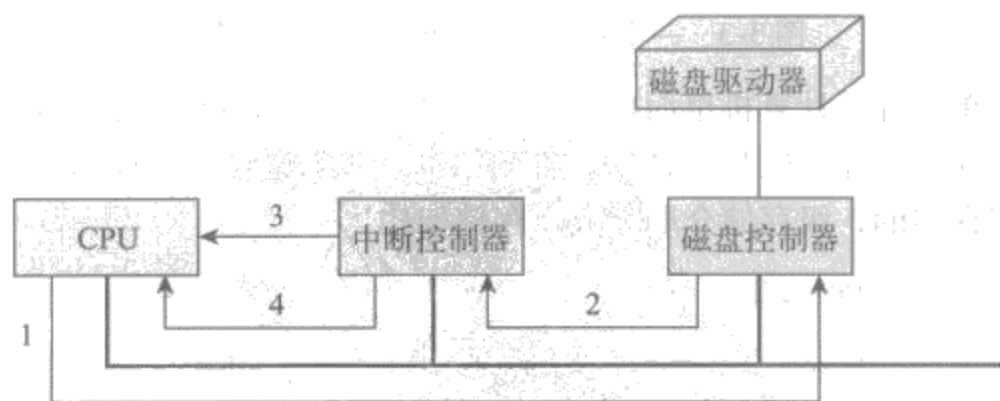


图 3-6 磁盘中断的响应流程

中断是很复杂的过程，中断处理过程中又可以发生中断，且还可以有所谓的软中断，即软件发出的中断。透彻理解中断对了解计算机操作系统的运行具有重要意义。因此，对中断机制不甚了解的读者请复习在计算机组成与体系结构课程中所学的中断内容。

3.2 抽象

我们已经多次提到过，操作系统提供的是一个抽象。所谓的抽象，就是在根本上存在但现实中不存在的东西。那么到底怎样理解抽象呢？

抽象来源于具体，但又超越具体。例如，人是具体的动物。但如果将人的具体属性，如肉体 and 骨架全部剥离，剩下的就是抽象，即人的灵魂。绘画史上有抽象派，而抽象画所表现的就是现实中不存在的东西，但这些东西确实又来源于现实，如瓦西里·康定斯基的“海战”（见图3-7）。

操作系统提供的抽象自然也来源于现实，就是具体的计算机硬件，CPU、内存、I/O 设备等。但又超出这些现实，给人提供了强于现实的东西，使人和应用软件感觉到更多、更好的硬件存在，而且只有在操作系统层面上，一般的人才会觉得计算机是可以使用的。

另外，抽象不光是操作系统提供给用户的一个存在，它也存在于操作系统内部。操作系统内部分为不同的功能块，而不同的功能块之间互相提供的也是抽象。

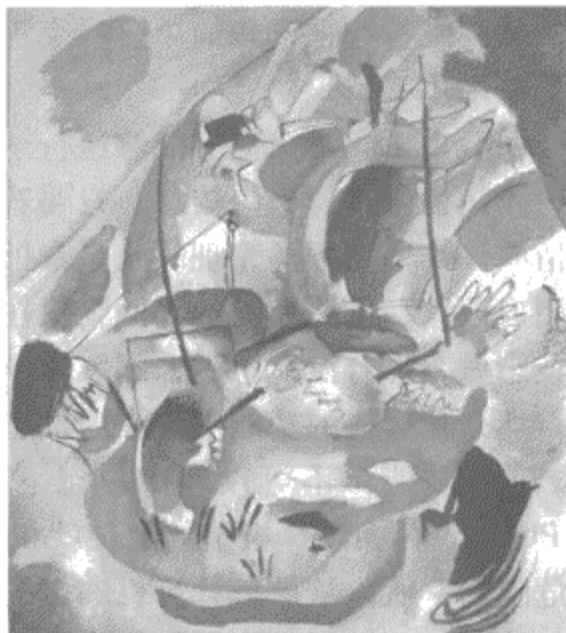


图 3-7 康定斯基的“海战”：
源于现实，又高于现实

3.3 内核态和用户态

就像世界上的人并不平等一样，并不是所有的程序都是平等的。世界上有的人占有资源多，有的人占有资源少，有的人来了，别人得让出资源，有的人则专门为别人让出资源。程序也是这样，有的程序可以访问计算机的任何资源，有的程序则只能访问非常受限的少量资源。而操作系统作为计算机的管理者，自然不能和被管理者享受一样的待遇。它应该享有更多的方便或特权。为了区分不同程序的不同权利，人们发明了内核态和用户态的概念。

那么什么是内核态，什么是用户态呢？只要想一想现实生活中，处于社会核心的人与处于社会边缘的人有什么区别就能明白处于核心的人拥有的资源多！因此，内核态就是拥有资源多的状态，或者说访问资源多的状态，我们也称之为特权态。相对来说，用户态就是非特权态，在此种状态下访问的资源将受到限制。如果一个程序运行在特权态，则该程序就可以访问计算机的任何资源，即它的资源访问权限不受限制。如果一个程序运行在用户态，则其资源需求将受到各种限制。

例如，如果要访问操作系统的内核数据结构，如进程表，则需要在特权态下才能办到。如果要访问用户程序里的数据，则在用户态下就可以了。

由于内核态的程序可以访问计算机的所有资源，这种程序的可靠性和安全性就显得十分重

要。试想如果一个不可靠的程序在内核态下修改了操作系统的各种内核数据结构，结果会怎样呢？整个系统有可能崩溃。而运行于用户态的程序就比较简单了，如果其可靠性和安全性出了问题，其造成的损失只不过是让用户程序崩溃，而操作系统将继续运行。

很显然，内核态和用户态各有优势：运行在内核态的程序可以访问的资源多，但可靠性、安全性要求高，维护管理都较复杂；用户态程序访问的资源受限，但可靠性、安全性要求低，自然编写维护起来都较简单。一个程序到底应该运行在内核态还是用户态取决于其对资源和效率的需求。

一般来说，一个程序能够运行于用户态，就应该让它运行在用户态。只在迫不得已的情况下，才让程序运行于内核态。只要看看一个国家的治理就清楚了。我们拿什么标准来判断什么事情应该归国家领导管理。凡是牵扯到计算机本体根本运行的事情都应该在内核态下执行，只与用户数据和应用相关的东西则放在用户态执行。另外，对时序要求特别高的事情，也应该在内核态做。你有没有想过，国家领导出门怎么不塞车呢？

那么什么样的功能应该在内核态下实现呢？首先，CPU 管理和内存管理都应该在内核态实现。这些功能可不可以在用户态下实现呢？当然能，但是不太安全。就像一个国家的军队（CPU 和内存存在计算机里的地位就相当于一个国家的军队的地位）交给老百姓来管一样，是非常危险的。所以从保障计算机安全的角度来说，CPU 和内存的管理必须在内核态实现。

诊断与测试程序也需要在内核态下实现。因为诊断和测试需要访问计算机的所有资源，否则怎么判断计算机是否正常呢？就像中医治病，必须把脉触摸病人。你不让中医触摸，他怎么能看病呢（当然，很多人认为中医是伪科学，根本治不了病，本书对此问题不做讨论）？输入输出管理也一样，因为要访问各种设备和底层数据结构，也必须在内核态实现。

对于文件系统来说，则可以一部分放在用户态，一部分放在内核态。文件系统本身的管理，即文件系统的宏数据部分的管理，必须放在内核态，不然任何人都可能破坏文件系统的结构；而用户数据的管理，则可以放在用户态。编译器、网络管理的部分功能、编辑器用户程序，自然都可以放在用户态下执行。图 3-8 描述的是 Windows 操作系统的内核态与用户态的界线。

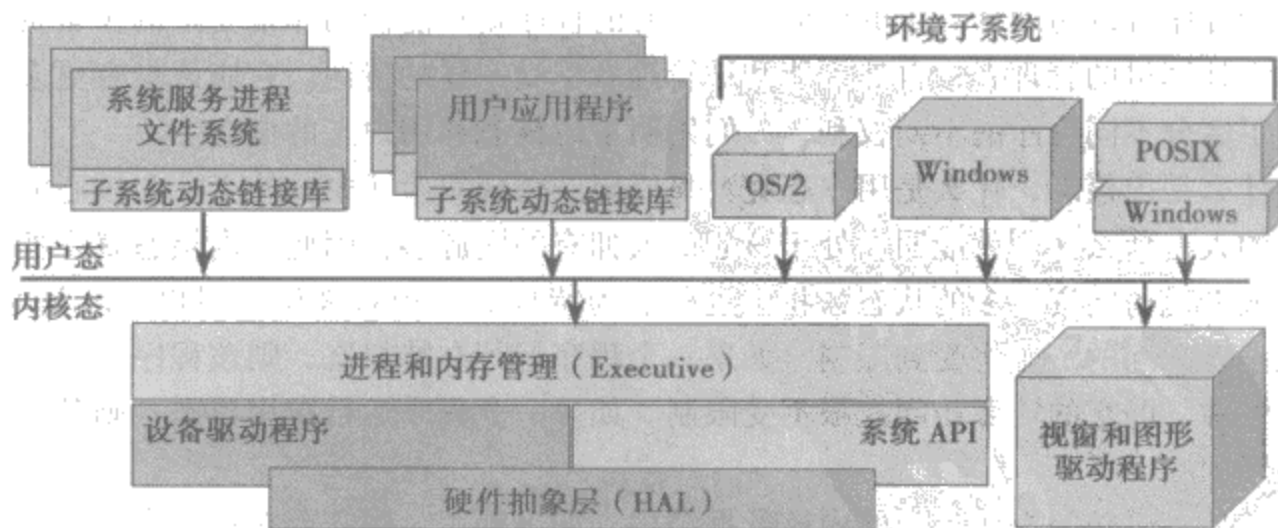


图 3-8 Windows 操作系统的内核态与用户态的界线

3.3.1 态势的识别

那么计算机是如何知道现在正在运转的程序是内核态程序呢？正确作出内核态或用户态的判断对系统的正确运行至关重要。显然作出这种判断需要某种标志。这个标志就是处理器的一个状态位。这个状态位是 CPU 状态字里面的一个字位。这就是说，所谓的用户态、内核态实际上是处理器的一种状态，而不是程序的状态。我们通过设置该状态字，可以将 CPU 设置为内核态，或者用户态，或者其他的子态（有的 CPU 有更多种子态）。一个程序运行时，CPU 是什么态，这个程序就运行在什么态。

3.3.2 内核态与用户态的实现

前面说过，内核态是特权态，而用户态是普通态。特权态下运行的程序可以访问任何资源，而用户态下的访问则受到限制。那么这种限制是如何实现的呢？

显然，要限制一个程序对资源的访问，需要对程序执行的每一条指令进行检查才能完成。而这种检查就是地址翻译。程序发出的每一条指令都要经过这个地址翻译过程。通过对翻译的控制，就可以限制程序对资源的访问。关于地址翻译的内容本书将在第 11 章详细阐述。

为了给内核态程序赋予访问所有资源的特权，系统处于内核态时，内核程序可以绕过内存地址翻译而直接执行特权指令，如停机指令。这种绕过翻译的做法突破了系统对资源的控制。

本书在讲完进程和内存后，将再次讨论内核态与用户态的议题。

3.4 操作系统结构

操作系统的结构也和操作系统历史类似，经历了好几个阶段。在操作系统刚刚出现时，人们还没有意识到操作系统的存在，也没有将那些库函数称为操作系统。那个时候，人们想到什么功能，就把这个功能加进来，并没有对所有这些功能进行统筹兼顾的计划。自然，那个时候的操作系统也是杂乱的、无结构的。

随着操作系统的进化，人们对操作系统的认识逐步加深，操作系统慢慢变得有一些结构了。各种功能归为不同的功能块，每个功能块相对独立，又经过固定的界面互相联系。任意一个功能块可以调用另一个功能块的服务。整个操作系统本身是一个巨大单一体（monolithic system），运行在内核态下，为用户提供服务，如图 3-9 所示。

后来人们发现单一体的操作系统结构有很多缺点：功能块之间的关系复杂，修改任意功能块将导致其他所有功能块都需要修改，从而导致操作系统设计开发的困难；这种没有层次关系的网状联系容易造成循环调用，形成死锁，从而导致操作系统可靠性降低。这时候，人们想到了人类社会里面的层次关系，何不将人类熟悉的层次关系搬到操作系统设计里来，给操作系统也定义个层次关系呢？将操作系统的功能分成不同层次，低层次的功能为紧邻其上的一个层次的功能提供服务，而高层次的功能又为更高一个层次的功能提供服务。就像人类团体里面的结

构：村长→镇长→县长→市长→……如图 3-10 所示。

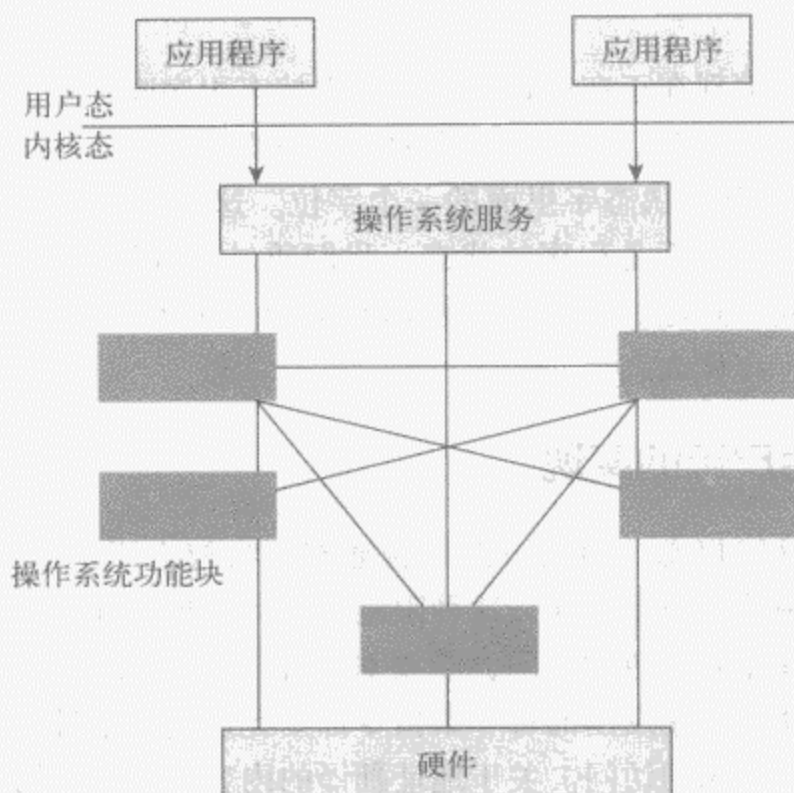


图 3-9 单一体的操作系统结构

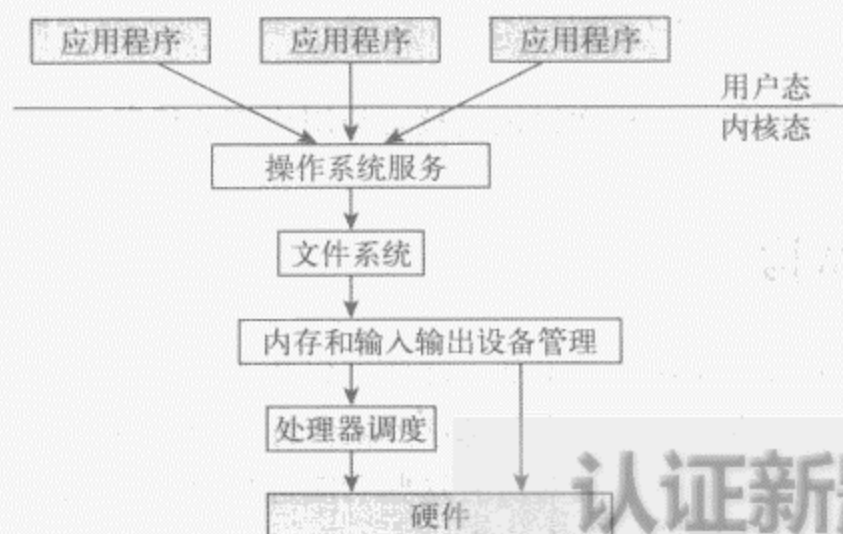


图 3-10 层次化的操作系统结构

从图 3-9 和图 3-10 可以看出，操作系统的所有功能都在内核态下运行。而这带来几个问题。首先，操作系统的所有服务都需要进入内核态才能使用，而从用户态转换为内核态是有时间成本的，这样就造成操作系统效率低下。在操作系统还比较简单时这个问题并不突出，但随着操作系统功能和复杂性的增加，这种问题就十分明显了。

其次，我们前面说过，在内核态运行的程序可以访问所有资源，因此其安全性和可靠性要求十分高。在操作系统很小时，将其设计得可靠和安全不是特别困难。再说，在操作系统历史的早期没有出现那么多的安全问题，自然安全上的考虑就不用太多。但随着操作系统越来越大，破坏者的水平越来越高，操作系统的可靠性和安全性就变得很难达到。只要想一想，1400 行的操作系统和 40 000 000 行的操作系统有什么区别就知道了。

因此，人们又想出了一个办法，就是微内核结构，即只将操作系统核心中的核心放在内核态运行，其他功能都移到用户态。这样就同时提高了效率和安全性（见图3-11）。

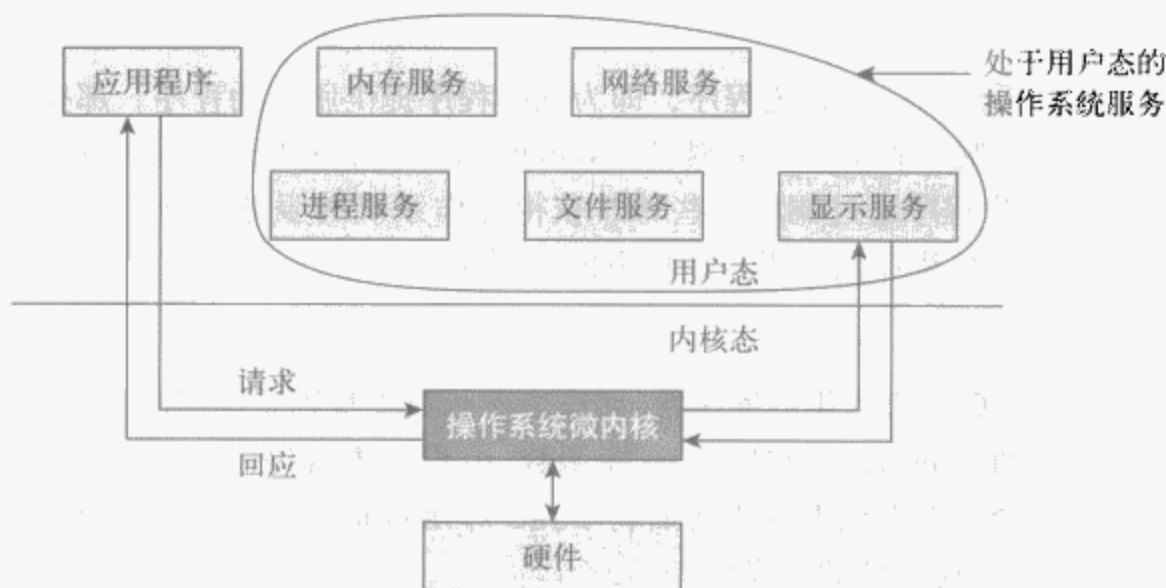


图3-11 微内核的操作系统结构

各种操作系统结构各有优缺点，但当前的趋势是第三种模式，即微内核的操作系统结构。至于这个微内核到底有多“微”，则是仁者见仁、智者见智的。例如，美国卡内基梅隆大学开发的 Mach 操作系统的内核非常小，而微软的 Windows XP 的内核就大多了。

3.5 进程、内存和文件

进程是操作系统里面的核心概念。它指的是一个运动中的程序。从名字上看，进程表示的就是进展中的程序。一个程序一旦在计算机里动起来，它就成为一个进程。操作系统对进程的管理通过进程表来实现。进程表里存放的是关于进程的一切信息。在任何一个时候，进程所占有的全部资源，包括分配给该进程的内存、内核数据结构和软资源形成一个进程核（Core）。核快照（Core Image）代表的是进程在某一特定时刻的状态。

如果在 Linux 或 UNIX 下编程序，当出现分段错误（segmentation fault）时，操作系统会自动进行核倒出（core dump）。“核倒出”把所有计算机的状态保存在一个文件中，通过阅读这个文件的内容可以得知出界时的进程状况，从而帮助对程序的调试。

进程与进程之间可以进行通信、同步、竞争，并在一定情况下可能形成死锁。这些概念都将在第4章进行详细的阐述。

内存是操作系统里面的另一个核心概念。它是进程的存放场所。如何对内存进行管理，使得数据的读写具有高效率、高安全、高空间利用率和位置透明的特性是内存管理所要达到的目的。

文件是操作系统提供的外部存储设备的抽象，它是程序和数据的最终存放地点。如何让用户的数据存放变得容易、方便、可靠和安全是文件系统要解决的问题。

3.6 系统调用

我们说过，操作系统是一个系统程序，即为别的程序提供服务的程序。那操作系统的服务是通过什么方式提供的呢？答案是系统调用（system call）。系统调用就是操作系统提供的应用程序界面（API）。用户程序通过调用这些 API 获得操作系统的服务。例如，如果用户程序需要进行读磁盘操作，在 C 程序代码里将使用下面的语句：

```
result = read(fd, buffer, nbytes);
```

这个 read 函数是 C 语言提供的库函数，而这个库函数本身则是调用的操作系统的 read 系统调用。注意这里有两个 read，一个是 read 库函数，由程序语言提供；一个是 read 系统调用，由操作系统提供。编译器在看到上述语句后将 read 库函数扩展为 read 系统调用。在真正执行时，操作系统将完成上述文件的读操作。

系统调用按照功能可以划分为六大类：

- 进程控制类。
- 文件管理类。
- 设备管理类。
- 内存管理类。
- 信息维护类。
- 通信类。

系统调用一般不在操作系统原理的课程中论述，而是在操作系统编程或系统编程的课程中论述。这里我们简单说一下系统调用的过程。系统调用分为三个阶段，分别是：

- 参数准备阶段。
- 系统调用识别阶段。
- 系统调用执行阶段。

在参数准备阶段，需要使用系统服务的程序将系统调用所需要的参数，如上述例子中的 fd, buffer, nbytes, 压到栈上。然后调用库函数 read。库函数 read 将系统调用 read 的代码放在一个约定好的寄存器里，通过陷入（trap，一种中断方式）将控制交给操作系统。由此进入到第二个阶段。操作系统获得控制后，将系统调用代码从寄存器里取出，与操作系统维护的一张系统调用表进行比较，获得系统调用 read 的程序体所在的内存地址。之后跳到该地址，进入到第三个阶段，执行系统调用函数。系统调用执行完毕后返回到用户程序（见图 3-12）。

系统调用中的参数传递

从图 3-12 可以看出，read 系统调用的参数压入到栈里面，即参数传递是通过栈来进行。但这并不是唯一的参数传递办法。事实上，这还不是效率最高的传递方法。效率最高的方法是将参数存放在指定的寄存器里面。由于寄存器的访问速度高于栈，这种参数传递将可以提升系统调用执行的效率。例如，在 x64 体系结构下，最前面的 8 个参数由寄存器传递。只有超过 8 个参

数时,超出的参数才通过栈来传递。

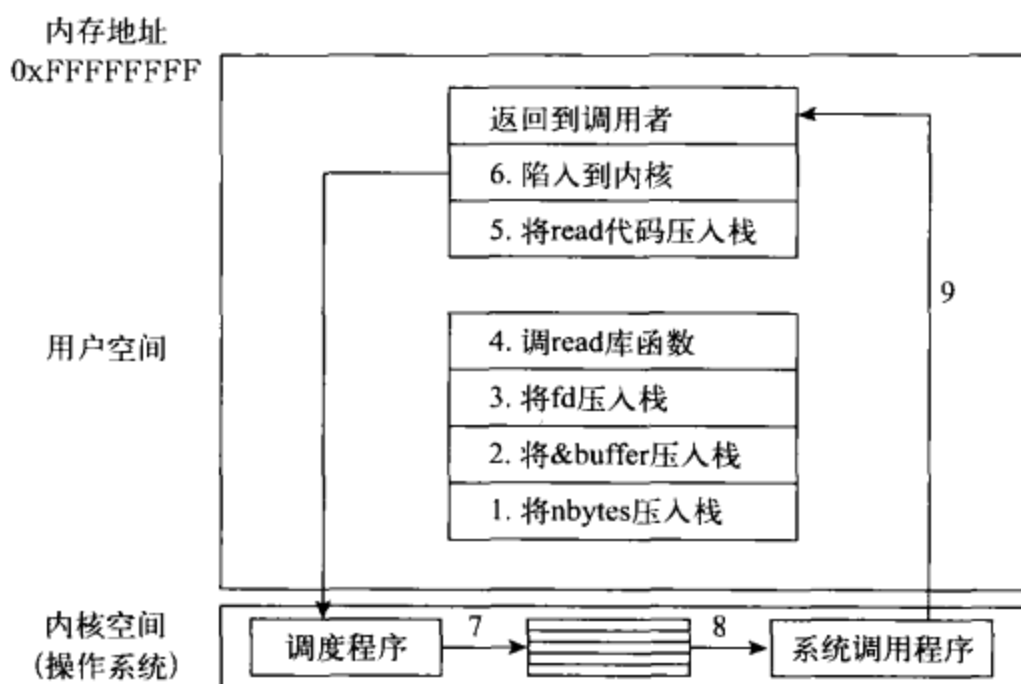


图 3-12 read 系统调用的过程

3.7 壳

前面一节说明了操作系统是如何给用户程序提供服务的。用户程序通过调用操作系统提供的系统调用 API 来获得操作系统的各种服务。但使用 API 需要编程。对于不编程序的用户来说,或对于需要与操作系统进行交互的用户来说,又怎么使用操作系统的服务呢?

为这些不编程的用户,操作系统提供了一个壳(shell),来与用户交互。每个操作系统都提供某种壳,以便与用户进行交互。这个壳是覆盖在操作系统服务上面的一个用户界面,既可以是图形界面,也可以是文本界面。用户在这个界面上输入命令,操作系统则执行这些命令。当然,用户输入的命令不是直接的操作系统服务,而是所谓的 utilities。utilities 的功用相当于 C 语言里面的库函数。因为用户不能直接调用系统调用(为什么不能呢?读者知道吗?),C 语言提供了库函数来解决这个问题。

同理,在壳上用户也不能直接使用操作系统的服务,而是通过 Utilities 来获得操作系统服务。UNIX 和 Linux 的壳都是文本形式,而 Windows 的壳是图形界面的。在 UNIX 和 Linux 里,要启动一个壳只需要运行 shell 即可。在 Windows 里面,要启动壳需要执行 explore.exe。在 Linux 和 UNIX 下可以同时启动多个壳,而在 Windows 下只能启用一个壳。2006 年,微软推出了 Powershell,从此改变了在 Windows 下只能启用一个壳的限制。Powershell 是一个文本命令壳,可以运行在 Windows XP SP2、Windows Server 2003、Windows Vista 和 Windows Server 2008 上。Powershell 提供了一种类似于 UNIX 和 Linux 上 utilities 的东西,称作 cmdlets。用户通过键入 cmdlets 里面不同的命令而指挥计算机进行各种操作。

一个壳的具体功能包括如下几项:

- 显示提示符,如 UNIX 下的提示符通常为 \$ 和 %。

- 接受用户命令并执行。
- 实现输入输出间接(或间接输入输出)。
- 启动后台进程。
- 进行工作控制。
- 提供伪终端服务。

例如:我们可以在 UNIX shell 上执行下述命令:

```
键入: $ date
显示: $ September 16, 2008
```

我们还可以进行输出间接(或间接输出):

```
键入: $ date >file
显示:
```

上述命令行里面的“>”符号是间接符合,将输出从显示屏转移到了文件 file。如果打开文件 file,里面的内容将是 September 16, 2008。

我们还可以同时进行输入和输出间接(或间接输入输出):

```
$ sort <file1 >file2
```

上述命令将 file1 里面的内容进行排序,然后将排序后的结果存放在文件 file2 里。

我们还可以将输出间接发送到打印机:

```
$ cat file1 file2 file3 > /dev/lp1
```

上述命令将 file1、file2、file3 里面的内容进行连接,然后将结果发送到打印机 lp1 上打印。

我们还可以启动后台程序:

```
$ make all >log &
```

上述命令启动一个编译后台程序,将编译的结果输出到文件 log 里。

那么壳是怎么实现的呢?下面是一个最为简单的壳,UNIX Shell:

```
while (TRUE) {                                /* 循环往复,以致无穷 */
    type_prompt();                             /* 显示命令提示符 */
    read_command(command, parameters)         /* 从用户获得命令 */
    if (fork() == 0) {                         /* fork 一个子进程 */
        execve(command, parameters, 0);       /* 执行用户命令 */
    }
    else {                                     /* 父进程代码段 */
        waitpid(-1, &status, 0);              /* 等待子进程结束 */
    }
}
```

这个壳一旦启动,就循环往复直到无穷。它所做的事情很简单:

- 1) 显示命令提示符。
- 2) 等待用户输入命令。

- 3) 使用 `fork` 创建一个子进程。
- 4) 使用 `execve` 在创建的子进程里执行用户输入的命令。
- 5) 重复步骤 1) ~ 4)。

上述程序片段里的 `fork` 和 `execve` 均为 UNIX 操作系统提供的系统调用。`fork` 的功能是创建一个子进程,并将自己的一切数据复制到子进程里,也就是说,`fork` 完成的实际上是自我复制。`execve` 的功能是用另外一个程序的内容覆盖自己,即执行新的程序。

这里值得一提的是,在上述程序代码里,`fork` 命令有两次返回:一次返回值为 0,表明是子进程(创建的新进程),对应 `if-else` 复合语句的 `if` 部分;另一次返回值不是 0,表明是父进程,对应 `if-else` 复合语句的 `else` 部分,而这个返回值就是子进程的进程 ID。也就是说,复合语句 `if-else` 的 `if` 和 `else` 部分在 `fork` 完成后都将执行。这听上去奇怪吗?

如果奇怪,那是因为你看到的是一个程序。我们知道,在一个程序里,`if-else` 语句的 `if` 和 `else` 两个部分只能有一个部分执行,而不能两个部分同时执行。我们还知道,一个函数是无法返回两次的。因为你只要写了 `return`,后面的语句就不能再执行了。那么 `fork` 命令为什么能返回两次呢?

这是因为 `fork` 的特殊功能使然。`fork` 的功效是创建一个和自己完全一样的进程。在 `fork` 系统调用完成后,我们面对的是两个进程,而不再是一个进程。这两个进程的程序代码完全一样,就是我们上面给出的代码。在其中一个进程中,执行的是 `if` 部分,在另一个进程中,执行的是 `else` 部分。因此,从每一个进程内部来看,复合语句 `if-else` 的语义并没有破坏。

而且,既然在 `fork` 后有了两个拷贝,那么每个拷贝里都有同样的 `fork` 调用语句。从另一个角度讲,有两个程序调用了 `fork`,这两次调用都需要返回。从内存结构看,这两个程序拷贝或者进程都有自己的调用堆栈,并都有调用返回的位置。对于每一个进程来说,它调用了一次 `fork`,获得了一次返回。因此,从每个进程内部看,函数返回一次的语义也得到遵守。

那么系统中的两个一模一样的进程如何区分彼此呢?简单,给每个进程一个不同的返回值即可。返回给父进程的是子进程的进程 ID,对应 `else` 部分;返回给子进程的是 0,对应 `if` 部分。

本章论述了操作系统的基本概念。这些概念是为下一步的进程、内存、文件和输入输出等章节的学习打下基础。在这些概念里面,最为关键的是内核态和用户态的定义极其关键。当然,抽象是操作系统的根本,而系统调用和操作系统结构也非常重要。

思考题

1. 操作系统提供的用户界面有几种? 分别是什么?
2. 举出几个人类社会采用的抽象,并与操作系统提供的抽象进行对比。
3. 有人说,内核态程序可以访问任何资源的权利对系统安全造成严重威胁,你怎么看?
4. 处理器的状态进行设置需要在何种态势下完成? 为什么?
5. 处理器从用户态转为内核态时面临的关键问题是什么? 如何解决?
6. 下述操作是否可以在用户态下执行?
 - a) 保护中断现场。

- b) 因系统调用陷入内核。
 - c) 启动外部设备。
 - d) 外设与内存经 DMA 进行数据交换。
 - e) 缺页中断。
7. 论述系统调用和壳之间的关系。
 8. 操作系统采用层次结构的优点是什么？层次结构有什么缺点吗？
 9. fork 是如何实现一次调用,两次返回的？它必要吗？为什么？
 10. 请调研当前工业界通用系统里面缓存、主存、磁盘、磁带的平均容量。
 11. 挑战题:内核态的特权是如何实现的？

第二篇 进程原理篇

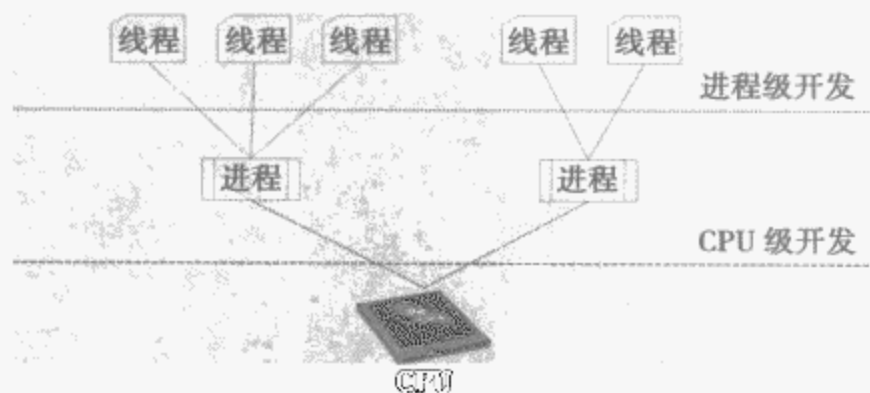
有了基础原理篇的铺垫,我们就可以进入到操作系统核心功能部件的讲解了。

计算机,顾名思义,是用来进行计算的,而进行计算的关键部件是计算机的芯片,即CPU。CPU能够按照一定的顺序进行正确计算是在一个指挥者的控制之下完成。这个指挥者就是操作系统。而操作系统对CPU进行管理和施行魔法的手段就是进程和线程。对进程和线程这些魔法道具进行阐述对理解操作系统自然十分重要,对其进行管理也就理所当然地成为操作系统的一个关键职责。本篇即对进程和线程进行详细讨论。

本篇内容繁多,包括第4章至第10章内容。第4章阐述的内容包括进程出现的逻辑必然性、多线程编程的效率、进程的创建和消亡、进程的状态及其转换、进程与地址空间、进程管理和进程模型的缺陷。第5章讲解的内容包括线程、线程管理、线程的用户态、内核态和混合态实现、现代操作系统的线程实现模型、多线程之间的关系、线程主要考虑的问题。第6章的内容包括为什么要通信、管道、记名管道、套接字、信号、信号量、共享内存、消息队列等。第7章的内容包括为什么同步、同步的目的、锁原语的进化、睡觉与叫醒原语、信号量、管程、消息传递和栅栏。第8章讲解的内容包括调度的目标、先来先服务、时间片轮转、短任务优先、优先级调度、混合调度、实时调度等算法,并对优先级倒挂和线程的不确定性进行讨论。第9章讲述如何使用中断启用和禁止、测试与设置来实现锁原语。第10章对死锁的产生、发展、防止与避免进行讲解,并讨论死锁、活锁和饥饿的关系。

进程和线程从根本上说是操作系统对CPU进行的抽象和装饰

本篇最重要的内容是并发。因为要并发,我们发明了进程,又进一步发明了线程。只不过进程和线程的并发层次不同:进程属于在处理器这一层上提供并发的抽象;线程则属于在进程这个层次上再提供一层并发的抽象。如果我们进入计算机体系结构里,就会发现,流水线提供的也是一种并发,不过是指令级并发。这样,流水线、线程、进程就从低到高在三个层次上提供我们所迫切需要的并发!



第4章 进 程

引子

在艾德蒙德哈雷的耐心劝说和敦促下,牛顿(见图4-1)著下了其传世名作《自然哲学的数学原理》(Philosophiae Naturalis Principia Mathematica,一部包含牛顿经典力学和万有引力定律的物理学著作)。在完成此书后,一种困惑始终挥之不去。牛顿写完书尾的总揽后对他的朋友哈雷感叹道:“一切物体的运动规律,天上的、地上的,我已经全部、完美地阐述清楚了。但一个一直困惑我的问题是:这些规律是从何而来的?或者说这些太空中的天体、地上的物体为什么会遵守这些规律呢?它们如果不遵守这些规律,难道我们还能把它们怎么样不成?”

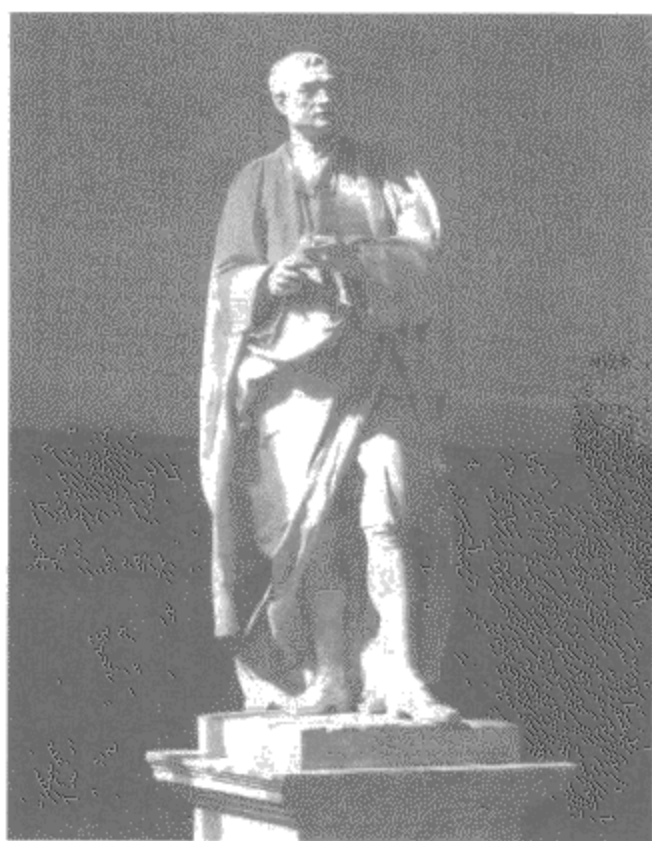


图4-1 位于剑桥大学三一学院门厅里的牛顿雕像

如果这些规律是随着时间与整个宇宙一起进化或演变的,则在进化到一半的时候,即在这些规律既是规律,又不是规律,或者说还是“半规律”时,宇宙是何以存在的呢?我们知道天体的运行轨道只要偏离一点,其存在就有可能大成问题,如果很多天体都随机运动,则后果将不堪设想。因此,按照“半规律”存在的宇宙是不可想象的。

剩下的唯一推论就是,这些规律在宇宙出现的一刹那就完美地存在了。或者更为直白地说,这些规律在宇宙存在之前就已经存在了。而这与约翰福音里的“太初有道(In the beginning was the word)”不谋而合。这里的 word 是希腊语原文 logos 的英文翻译。而 logos 就是逻辑,就是规律。这句话的意思是在还没有宇宙的时候,规律就已经存在了。

而由于规律本身不是物质,其存在自然是无中生有了。至于是谁或者什么导致这些规律的存在,那就留给读者自己去遐思了……

本章讨论的进程也是无中生有。这点与宇宙的运行规律有一些相似。不过与宇宙规律不同的是,我们知道是谁导致了计算机中进程的出现:是人!

当人们面临困境时通常的做法就是:发明新的概念、新的术语或新的机制来解脱困境。

4.1 进程概论

进程管理、内存管理和文件管理是操作系统的三大核心功能。那么什么是进程呢?顾名思义,进程就是进展中的程序,或者说进程是执行中的程序。就是说,一个程序加载到内存后就变为进程。即:

$$\text{进程} = \text{程序} + \text{执行}$$

进程在 Multics 操作系统出现前叫做工作(job)。“工作”是 IBM 用于多道批处理程序设计中的概念。由于历史的原因,Multics 操作系统的研发人员不愿意承用 IBM 发明的术语,将工作改为了进程(process)。那么进程出现的动机是什么呢?

本书在第2章说过,单一操作员单一控制终端、批处理均存在效率低下的问题,即 CPU 使用率不高。为了提高 CPU 利用率,人们想起将多个程序同时加载到计算机里,并发执行。这些同时存在于计算机内存的程序就称为进程。进程让每个用户感觉到自己独占 CPU。因此,进程就是为了在 CPU 上实现多道编程而出现的概念,如图 4-2 所示。

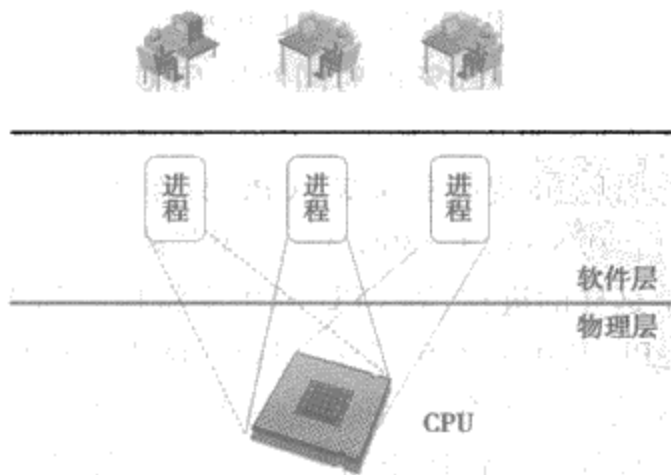


图 4-2 进程让每个用户感觉到自己独占 CPU

4.2 进程模型

那么进程到底是个什么东西呢？什么是进展中的程序呢？从物理内存的分配来看，每个进程占用一片内存空间，从这点上说，进程就是内存的某片空间。由于在任意时刻，CPU 上只能执行一条指令，所以任意时刻上在 CPU 上执行的进程只有一个，而到底执行哪条指令由物理程序计数器指定。也就是说，在物理层面上，所有进程共用一个程序计数器。

而从逻辑上来看，每个进程可以执行，也可以暂时挂起让别的进程执行，之后又可以接着执行。这样，进程就需要某种办法记住每次挂起时自己所处的执行位置，这样才能在下次接着执行时从正确的地点开始。因此，从这个角度看，每个进程有着自己的计数器，记录自己下条指令所在的位置。从逻辑上说，程序计数器可以有很多个。

而从时间上看，每个进程都必须往前推进。在运行一定的时间后，进程都应该完成了一定的工作量，即每次进程返回，它都处在上次返回点之后。这就像古希腊哲学家赫拉克里特说过的“一个人不能两次踏入同一条河流。”进程的这三种概念可以由图 4-3 表示。

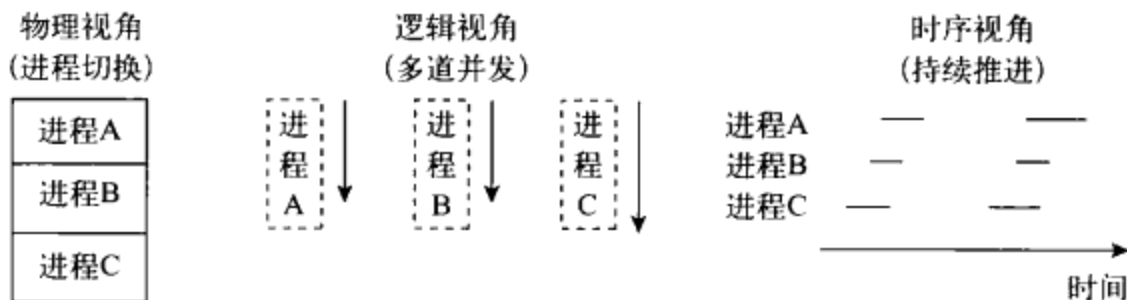


图 4-3 进程模型的三个视角

这里需注意的是，进程不一定必须终结。事实上，许多系统进程（用来为别的进程提供系统服务的进程）是不会终结的，除非强制终止或计算机关机。

4.3 多道编程的效率

我们发明进程是为了多道编程，而多道编程的目的则是提高计算机 CPU 的效率，或者说是系统的吞吐量。例如，如果一个进程用 20% 的时间使用 CPU 进行计算，另外 80% 的时间用来进行 I/O，则如果使用单道编程，CPU 的利用率只有 20%。但如果同时运行两个这样的进程，即进行所谓的 2 道编程，则 CPU 利用率将提高到 36%（CPU 只在两个进程同时进行 I/O 时才处于闲置状态，因此 CPU 利用率为 $1 - 0.8 \times 0.8 = 36\%$ ）。这里忽略了进程切换所需要的系统消耗。

同理，如果同时运行 3 个这样的进程，则 CPU 利用率将提高到 48.8%。4 个进程的 CPU 利用率将为 59%，5 个进程的 CPU 利用率为 67.2%。这样，随着进程数量的增加，也就是随着多道编程的度的增加，CPU 利用率将逐步提升，直到某个临界点时为止。这个临界点就是多道编程的极限。超过这个极限，多道编程的好处将逐步消失，甚至呈下降趋势。对于我们这个系统来说，多道编程的度达到 6 以后，CPU 利用率的提升就很小了，而进程切换所带来的系统消耗则变得明

显。多道编程的度与 CPU 效率的关系,如图 4-4 所示。

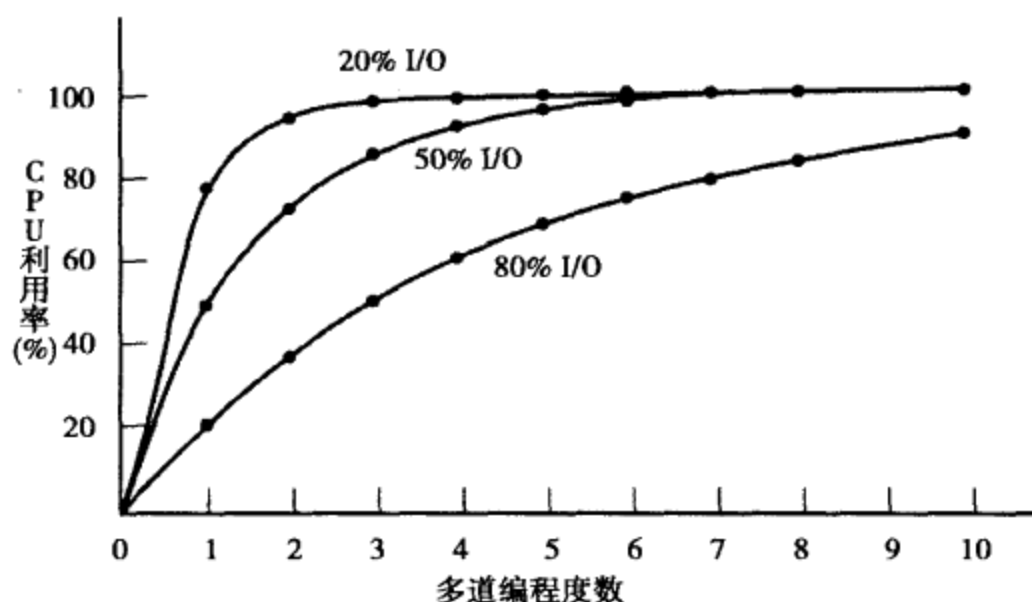


图 4-4 多道编程度数、I/O 时间和 CPU 利用率的关系(来源:参考文献[3])

下面我们通过一个多道编程的具体例子,来看看多道编程时计算机里面事件的发生顺序和多道编程环境下系统响应时间的提升。

假定我们有 4 个程序,每个程序花费 80% 的时间进行 I/O,20% 的时间使用 CPU。每个程序的启动时间和其需要使用 CPU 进行计算的分钟数如表 4-1 所示。

表 4-1 4 个程序的启动时间和所需 CPU 时间

程序编号	启动时间	需 CPU 时间(分钟)
1	00:00	4
2	00:10	3
3	00:15	2
4	00:20	2

下面我们看看该计算机里面事件的发生顺序。

从 0 点 0 分开始到 0 点 10 分,系统里只有 1 个程序,因此属于单道编程状态。单道编程时 CPU 的利用率为 20%,因此第 1 个程序在该 10 分钟里总共使用了 CPU 达 2 分钟(其他 8 分钟都用来进行 I/O 了)。0 点 10 到 0 点 15 分钟,系统里有两个程序,因此属于 2 道编程。我们前面计算过,2 道编程时 CPU 利用率为 36%,则在该 5 分钟时间内,CPU 使用了 1.8 分钟。假定这两个程序完全平等,则每个程序使用 CPU 的时间为 0.9 分钟。至此,程序 1 总共运行了 2.9 分钟 CPU 时间,程序 2 运行了 0.9 分钟 CPU 时间。

从 0 点 15 分开始到 0 点 20 分,系统里有 3 个程序,因此属于 3 道编程状态。3 道编程时 CPU 的利用率为 48.8%,则在这 5 分钟时间内,CPU 被占用了大约 2.4 分钟(其他 2.6 分钟都用来进行 I/O 了)。同样,假定所有程序完全平等,则每个程序使用 CPU 的时间为 0.8 分钟。至此,程序 1 总共运行了 3.7 分钟 CPU 时间,程序 2 运行了 1.7 分钟 CPU 时间,程序 3 运行了 0.8 分钟 CPU 时间。此时,程序 1 离结束所需要的 CPU 时间最短,仅为 0.3 分钟。

从 0 点 20 分开始,系统里有 4 个程序,因此属于 4 道编程。我们前面计算过,4 道编程时 CPU 利用率为 59%。而如果程序 1 想再运行 0.3 分钟 CPU 时间,则整个系统需运行时间约为 2 分钟(2 分钟时间内 CPU 共被使用 1.2 分钟,平均每个程序使用 CPU 时间为 0.3 分钟)。因此,在 0 点 22 分时,第 1 个程序执行完毕,系统变为 3 道编程。

此时,程序 1 结束,程序 2 总共运行了 2 分钟 CPU 时间,程序 3 运行了 1.1 分钟 CPU 时间,程序 4 则运行了 0.3 分钟 CPU 时间。此时,程序 3 离结束所需的 CPU 时间最短,为 0.9 分钟。那么系统需要运行多长时间才能使程序 3 获得 0.9 分钟的 CPU 时间呢? 答案是 5.6 分钟。因为 3 道编程的 CPU 利用率大约为 48%,而 5.6 分钟内 CPU 的使用时间约是 2.7 分钟。三个程序平分,每个程序运行了 0.9 分钟 CPU 时间。因此,到 0 点 27.6 分钟时,系统里只剩下两个程序。而在 1.6 分钟后,即 0 点 28.2 分钟时,程序 2 将结束运行,剩下程序 4 一个程序。而该程序则在 0 点 31.7 分钟时结束运行。整个事件发生顺序,如图 4-5 所示。

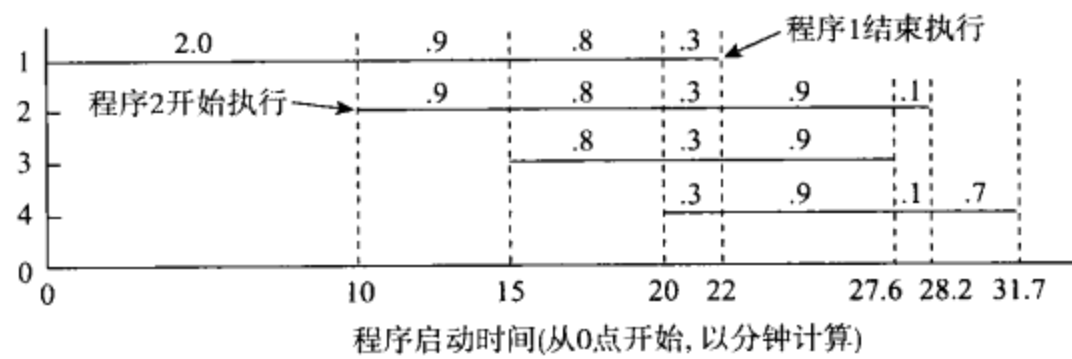


图 4-5 多道编程时事件的发生顺序 (来源:参考文献 [3])

在多道编程环境下, 4 个程序和整个系统的响应时间为:

程序	程序 1	程序 2	程序 3	程序 4
响应时间	22 分钟	18.2 分钟	12.6 分钟	11.7 分钟
系统平均响应时间	16.125 分钟			

而在单道编程环境下, 4 个程序和整个系统的响应时间为:

程序	程序 1	程序 2	程序 3	程序 4
响应时间	20 分钟	25 分钟	30 分钟	35 分钟
系统平均响应时间	27.5 分钟			

由上述两个表格的数据看出, 多道编程比起单道编程, 系统平时响应时间缩短了 11.375 分钟, 响应时间减少了 41.37%。由此可见多道编程的巨大好处。当然了, 多道编程带来的好处到底有多少与每个程序的性质、多道编程的度数、进程切换消耗等均有关系。但一般说来, 只要度数适当, 多道编程总是利大于弊。

4.4 进程的产生与消失

什么事件可以造成进程的产生和消亡呢? 当然有很多这样的事件。对于进程产生来说, 主要的事件有:

- 系统初始化 (神创造人)。
- 执行进程创立程序 (人生子)。

- 用户请求创立新进程（试管婴儿）。

在一个系统初始化时，将有许多进程产生。这些产生的进程是系统正常运行必不可少的。这些进程的存在使得新的进程和用户程序的执行成为可能。例如：在系统初始化后，Windows 操作系统将自动产生诸如对话管理（SMSS）、登陆管理（WINLOGON）、安全管理（LSASS）、Windows 子系统（CSRSS）、Windows 壳（explore）等系统进程。

在系统初始化后，系统就等待用户输入命令。如果这个用户启动一个程序，如双击一个可执行文件，那么系统将为这个可执行文件创立一个进程。除此之外，用户也可以在程序里面通过系统调用，如 `fork` 或者 `CreateProcess` 直接生成新的进程。

造成进程消亡的事件则可以分为四种情况：

- 寿终：进程运行完成而退出。
- 自杀：进程因错误而自行退出。
- 他杀：进程被其他进程强行“杀死”。
- 处决：进程因异常而强行终结。

前面两种情况均为自愿退出，后面两种情况均为非自愿退出。在程序设计时我们追求的是前面两种退出，也算是我们在虚拟世界里面追求人权（进程权）的努力吧。第3种情况通常是一个父进程发出命令“杀死”一个子进程。当然，一个用户也可以“杀死”自己的进程，但不能“杀死”别人的进程。但一个超级用户（具有系统管理员特区）则可以“杀死”任何进程。第4种情况在一个进程进行某种非法操作，如访问出界或者除以0之后发生，而这种非法操作将被操作系统捕捉。操作系统捕捉到这种异常后将终结造成异常的进程。

4.5 进程的层次结构

我们说过，一个进程在执行过程中可以通过系统调用创建新的进程。这个新创建出来的进程就称为子进程，创建子进程的进程则称为父进程。子进程又可以再创建子进程，这样子子孙孙创建下去就形成了所谓的进程树。UNIX 称这个进程树里面的所有进程为一个进程组，进程组里面的进程分布在不同的层次上，从而形成一个层次架构。

Windows 没有进程组的概念，而是所有进程均地位平等。

4.6 进程的状态

我们前面说过，进程可以在 CPU 上执行，也可以处于挂起状态。显然，一个进程至少有这么两个状态。那么进程还有别的状态吗？

如果进程在 CPU 上执行，自然就是执行状态。而如果是挂起状态呢？那就得看是什么原因挂起的。因为操作系统在进行进程调度时要从挂起的进程里面选择一个来执行，所以清楚一个进程挂起的原因对调度的有效推进十分重要。

那么进程挂起有什么原因呢？首先是一个进程在运行过程中执行了某种阻塞操作，如读写

磁盘。由于阻塞操作需要等待结果后才能继续执行,操作系统将把这个进程挂起来,让其他进程运转。另外一种情况是一个进程执行时间太长了,为了公平,操作系统将其挂起,让其他进程也有机会执行。

这两种挂起的原因十分不同:第一种挂起是进程自身的原因。这个时候,即使我们把 CPU 控制权交给它,它也无法运行。第二种挂起是操作系统的原因。进程自己并无问题。只要把 CPU 交给进程,它就可以立即运行。这样,如果我们将挂起进程分为这样两类,操作系统在进程调度时就只需要查看第二类进程,而无需浪费时间查看第一类进程。

因此,我们将进程分为三种状态:执行、阻塞和就绪,如图 4-6 所示。

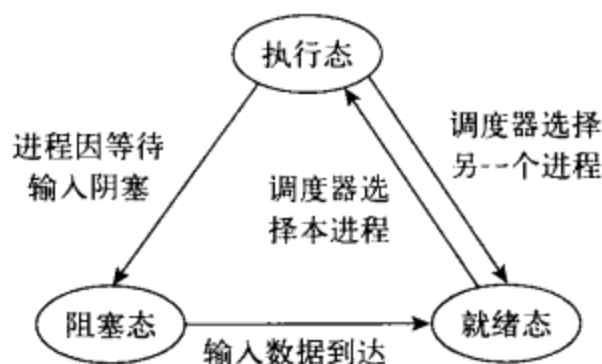


图 4-6 进程的三个典型状态

三个状态之间可以进行各种转换。如果每个状态都可以转换为另外一个状态,则一共有 6 种转换:

- 执行→就绪
- 执行→阻塞
- 阻塞→就绪
- 就绪→执行
- 阻塞→执行
- 就绪→阻塞

问题是,上面的转换并不是都可以发生。一个进程在执行时,因为运行时间太长,操作系统可以将其挂起,转换为就绪状态。因此第 1 种转换是可以的。进程执行过程中如果执行了某种阻塞操作,则将进入阻塞状态。因此第 2 种转换也是可以的。一个阻塞的进程在其等待的资源达到后,就可以随时执行,成为就绪状态,因此第 3 种转换也是可以的。就绪进程由操作系统调度到 CPU 上就成为执行状态,因此第 4 种转换也是可以的。

但是第 5、第 6 两种转换却是不可以的。我们前面讲过,阻塞进程即使被给予 CPU,也无法执行,操作系统在调度时并不会在阻塞队列里挑选。因此,阻塞状态无法转换为执行状态。对于处于就绪状态的进程来说,它因为并没有执行,自然无法进入到阻塞状态。这就像一个人不往前走,自然就不会有任何人是其障碍。因此,就绪状态无法转换为阻塞状态。

这里需要注意的是,第 5、第 6 两种转换虽然都不存在,但其原因是两样的。第 5 种转换不是因为我们不让它发生。如果我们乐意,完全可以让操作系统在阻塞队列里挑选一个进程予以执行,只不过这个进程在执行第 1 条指令时就会又发生阻塞(因为其等待的数据尚不可用或者发生异常)。因此,从理论上说,阻塞到执行是可以的,只不过这种状态转换没有任何实际价值而被操

作系统禁止。(就像每个人都可以给自己扇耳光,但我们奉劝各位不要这样做,因为没有什么益处)。而第6种状态转换则在理论上不可以。一个进程只能在执行时才可能阻塞,没有执行的进程无法直接转换到阻塞状态。

这里阐述的进程三状态并不是唯一的分类方式。事实上,许多商业操作系统的进程状态不止三个,例如,Windows的进程有7个,Solaris里面的进程则有6个。但不管3个、6个、7个还是几个,其目的都是便于操作系统对进程的管理。只要细分对管理有利,我们就细分。否则就维持三状态。

4.7 进程创立

进程创立步骤:

- 1) 分配进程控制块。
- 2) 初始化机器寄存器。
- 3) 初始化页表。
- 4) 将程序代码从磁盘读进内存。
- 5) 将处理器状态设置为“用户态”。
- 6) 跳转到程序的起始地址(设置程序计数器)。

这里一个最大的问题是跳转指令是内核态指令,而在第5步时处理器状态已经被设置为用户态,而用户态下是不能执行内核态指令的。这个问题是怎么解决的呢?当然了,这就需要硬件帮忙了。硬件必须将第5、第6两步作为一个步骤一起完成。

进程创立在不同的操作系统所需的方法也不一样。例如,UNIX将进程创立分作两个步骤:第1个步骤是fork,创建一个与自己完全一样的新进程;第2个步骤是exec,将新的进程的地址空间用另一个程序的内容覆盖,然后跳转到新程序的起始地址,从而完成新程序的启动。而Windows使用一个系统调用就完成进程创建。这个系统调用就是CreateProcess。在调用该函数时我们把欲执行的程序名称作为参数传过来,创建新的页表,不需要复制别的进程。

UNIX和Windows的进程创建过程各有优缺点。UNIX的创建过程要灵活一些,因为我们既可以自我复制,也可以启动新的程序。而自我复制在很多情况下是很有用的。例如,Web服务器在每收到一个用户请求后,就创建一个新的一摸一样的进程来服务用户请求。而在Windows下,复制自我的过程就要复杂一些。而且,共享数据只能通过参数传递来实现。

4.8 进程与地址空间

进程空间也称为地址空间。地址空间就是进程要用的所有资源。所有资源构成了状态的划分。不可能有两个进程状态完全一样。所以每个进程对应计算机的一种状态,而计算机状态就是所有存储单元的内容。

地址空间的特点就是“被动”,自己不能做什么,只提供支持。打个比方。看过演出吗?话剧、芭蕾、歌剧、京剧?有个舞台,那些道具和舞台就是地址空间。这些空间本身不能发生任何动

作,做动作的只能是演员。而那些演员就是我们将要讲述的线程。跳上来一个演员就是一个线程,如图4-7所示。



图4-7 音乐剧《剧院魅影》(The Phantom of the Opera):
舞台布景及设施是地址空间,而舞台上男女演员是线程

进程与地址空间研究的主要内容是如何让多个进程空间共享一个物理内存。具体来说,就是高效、安全地让所有进程共享这片物理内存。就像一个政府要完成的任务就是使其国家的人民平等(至少是口头上)地共享这片国土。

4.9 进程管理

那么谁管理进程的资源?操作系统。本书前面说过,Operating的意思就是掌控一切。那么怎么掌控呢?操作系统要掌控一切状态,就必须拥有某些手段或资源。那需要什么手段或资源呢?如果让你监视一群人,要你掌握他们的一切情况,你第一件要做的事是什么?装监视器?不是!而是要知道这群人到底是哪些人!即你需要知道并维持这群人的各种信息!

4.9.1 进程管理所需要的手段

本章前面有一节讲到进程的产生和终结,但这到底是什么意思呢?产生一个进程对于操作系统来说意味着什么呢?进程消亡又对操作系统有何影响呢?要回答这个问题只需要看一下一个人出生对一个社会来说意味着什么就可以了。在一个人出生后,医院需要在几天内为其建立记录,该记录包括诸如姓名、性别、体重、身高、父母为何人、在何时何地出生、健康状态等信息,然后该记录用来登记户口、办理身份证等。在这些手续之后,这个人就正式存在了。有了这些记录,政府就可以对这个人进行各种管理了。

与一个社会管理人类类似,操作系统要管理进程就要维护关于进程的一些信息。当一个进程产生时,操作系统也需要为其建立记录。而操作系统用于维护进程记录的结构就是进程表或进程控制块(PCB, Process Control Block)。这个进程表或PCB里面存放的就是有关该进程的资料。那么进程表里有什么资料呢?显然,不同的操作系统维护的进程资料是不尽相同的。但一般说

来,维护的资料信息应当包括寄存器、程序计数器、状态字、栈指针、优先级、进程 ID、信号、创立时间、所耗 CPU 时间、当前持有的各种句柄等。而采纳的数据结构主要是线性表、链表和结构 (Struct),当然也可能使用树和图(网络)结构。例如,Solaris 的进程表就使用了上述 4 种数据结构。表 4-2 描述的是一个极度简化了的进程表。

表 4-2 进程表所包含的进程记录信息

进程基本信息指针(进程 ID、创建用户 ID、创立时间等)
进程家族树指针(子进程、父进程、孙子进程、祖父进程)
进程资源信息指针(寄存器、栈指针、当前持有句柄等)
信号支持
进程状态信息指针(程序寄存器、状态字、优先级等)
时间统计信息(所占 CPU 时间、子进程所占 CPU 时间等)
其他需要的信息指针……
其他需要的信息指针……

而这个进程表保持在操作系统所在的内核空间里,如图 4-8 所示。

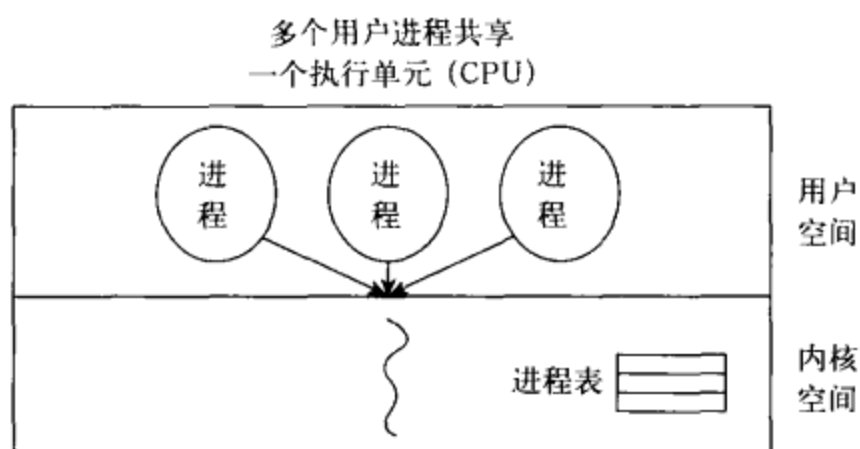


图 4-8 进程表存放在操作系统所在的内核空间里

如果想更深入地了解进程表的结构,读者可以参阅相关商业操作系统的内核教程。

4.9.2 进程管理要处理的问题

进程管理的最大问题是资源分配。那么怎么分配资源?人类社会最大的问题也是分配资源。谁能解决资源争端,让地球上每个人高高兴兴地共享资源,谁就是人类的救星。当然,计算机的资源分配问题不像人类那样困难,因为程序没有自我意识,就算我们对某些程序不公平,它们也无法抱怨。但这不能成为我们偷懒和堕落的理由。毕竟,我们制造计算机就是想好好地利用它,自然希望能够让所有进程高兴地处在一起。再说了,我们的本性还是追求公平的。

除了公平之外,还有一个问题要考虑:效率。也就是最优。每个进程分配同样的资源不行,大锅饭不行。以前 32 个终端连到一台计算机上,慢的不行,结果没有任何人高兴。不如让部分人先富起来,给他们使用资源的优先权。

这样,公平与效率,就成了进程管理中永恒的主题。到底是公平重要?还是效率重要?天平的不同倾斜将引出十分不同的进程管理模式。

4.10 进程的缺陷

看上去,进程是个很好的东西。它提供多道编程,让我们感觉我们每个人都拥有自己的 CPU 和其他资源,可以提高计算机的利用率,可以让我们练习如何实现公平和效率。

那么进程有什么问题吗?

如果善于观察,就会发现,进程有个很严重的问题。假定现在有两部很好的电影,都只放一次,以后再也不放了。而且,这两部电影同时放,当然了,是在不同的两个房间放。而你很想将这两部电影都看了,有什么办法吗?假定没有光碟刻录机也没有录像机等。

当然,我们没有办法同时看两部电影。这也是进程的缺点。它只能在一个时间干一件事情。如果想同时干两件或多件事情,进程就不够用了。

另外,更为重要的是,进程在执行过程中如果阻塞,例如等待输入,整个进程就将挂起(暂停),而无法继续执行。这样,即使进程里面有部分工作不依赖于输入数据,也无法推进。

为了解决上述两个问题,人们就发明了线程。

思考题

1. 发明进程的根本动机为何?它与程序是什么关系?请予以论述。
2. 进程带给我们的最大好处是什么?它有什么缺点吗?
3. 进程空间是什么意思?它包括哪些东西?它与进程是什么关系?
4. 进程有哪 3 种状态,分别代表什么意思?
5. 进程 6 种状态转换里有两种是不存在的,但它们不存在的理由却不一样。请予以解释。
6. 在实际商用操作系统里,进程的状态通常多于 3 个,请问,设置多于 3 个进程状态的可能原因是什么?
7. 操作系统管理进程的根本手段是什么?
8. 进程管理时的两个重要考虑是公平和效率。除此之外,还有什么因素需要考虑吗?
9. 进程的产生与消亡与人的出生与消亡有着某种类比性,你能否予以阐述?
10. 多道编程是否总是能提高 CPU 的利用效率?为什么?
11. 有同学认为进程状态的 6 种转换中,就绪到阻塞在理论上可以(操作系统将就绪进程的状态改变为阻塞状态),而从阻塞到运行则在理论上不可能,你同意吗?为什么?
12. 分析:在内核态下的进程通常都共享一个地址空间,这是为什么?

第5章 线 程

引子

公元前337年,年仅20岁的亚历山大继位成为马其顿(Macedonia)国王。此时,相邻的希腊各城邦共和国纷纷起来反叛马其顿的统治。北方的野蛮民族也蠢蠢欲动。亚历山大的侍臣顾问们纷纷建议亚历山大放弃对希腊各城邦共和国的平叛,而集中精力对北方野蛮种族进行毁灭性打击。这些侍臣顾问们认为,亚历山大一个人无法分身分神同时对付希腊各城邦和北方各野蛮种族,因此,只能选择其一进行打击,通过杀鸡给猴看让另外的人臣服。

但是亚历山大拒绝了侍臣顾问们的建议。他认为,只要他速度足够快,就能够完成先破希腊再定北方边界的分身术。亚历山大的神速决战,迅速击败了希腊各城邦。而此时,北方野蛮种族还没有回过神来,当亚历山大出现在他们面前时,他们行动的时机已经失去了。

而就是这种神速的时间分身术,让亚历山大在一继位就慑服了所有闻其名的人,也奠定了其后来东攻西掠、南征北战、征服整个文明世界的壮举……,如图5-1所示。



图5-1 雕塑:亚历山大在战斗中

每个人在人生的某个时候,都希望自己能够分身,从而完成某件不可能完成的事。而进程也是一样,它也希望在某些时候能够分身,从而完成更加复杂的使命。于是便有了线程。

5.1 进程的分身术——线程

那么线程是什么?我们知道,进程是运转的程序,是为了在 CPU 上实现多道编程而发明的一个概念。但是进程在一个时间只能干一件事情。如果想同时干两件事,例如同时看两场电影,我们自然想到传说中的分身术,就像孙悟空那样同时变出多个真身。

当然,人在现实中进行分身是办不到的。但进程却可以办到,办法就是线程。线程就是我们为了让一个进程能够同时干多件事情而发明的“分身术”。

既然线程是进程的分身,每个线程自然在本质上是一样的,即拥有同样的程序文本。但由于是分身,自然也应该有不一样的地方,这就是线程执行时的上下文不一致。事实上,我们说线程是进程里面的一个执行上下文,或者执行序列。显然,一个进程可以同时拥有多个执行序列。这就像舞台,舞台上可以有多个演员同时出场,而这些演员和舞台就构成了一出戏。类比进程和线程,每个演员是一个线程,舞台是地址空间,这个同一个地址空间里面的所有线程就构成了进程。

在线程模式下,一个进程至少有一个线程,但也可以有多个线程,如图 5-2 所示。

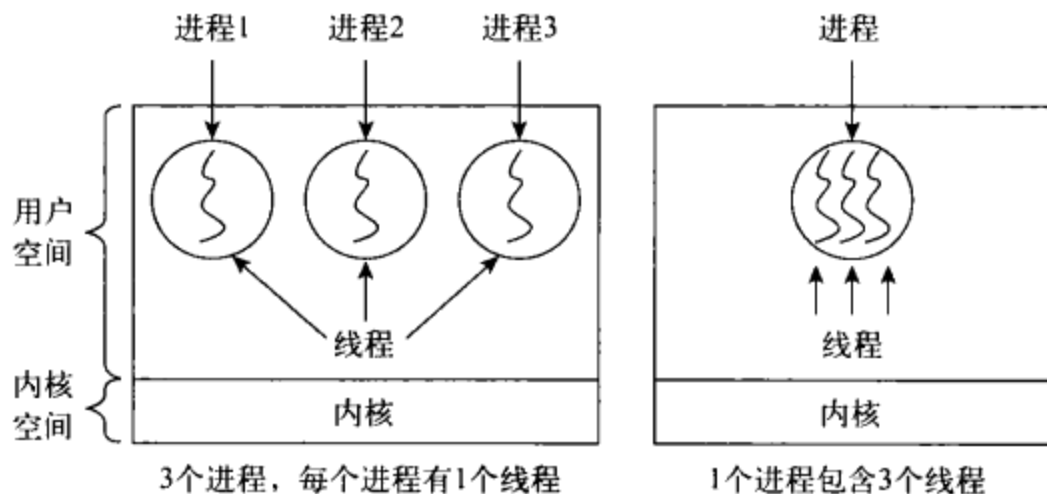


图 5-2 线程模型下的单线程进程和多线程进程对比

将进程分解为线程还可以有效利用多处理器和多核计算机。在没有线程的情况下,增加一个处理器并不能让一个进程的执行速度提高。但如果分解为多个线程,则可以让不同的线程同时运转在不同的处理器上,从而提高了进程的执行速度。例如,当我们使用文字处理软件,如 Microsoft Word 时,实际上是打开了多个线程。这些线程一个负责显示,一个接受输入,一个定时进行存盘。这些线程一起运转,让我们感觉到我们的输入和屏幕显示同时发生,而不用键入一些字符,等待一会儿才看到屏幕显示。在我们不经意间,文字处理软件还能自动存盘。当然,此项操作取决于系统当时的状况,有时我们会感觉到存盘时,计算机接受输入的速度慢了下来。但在绝大部分情况下,一切都还是令人满意的,如图 5-3 所示。

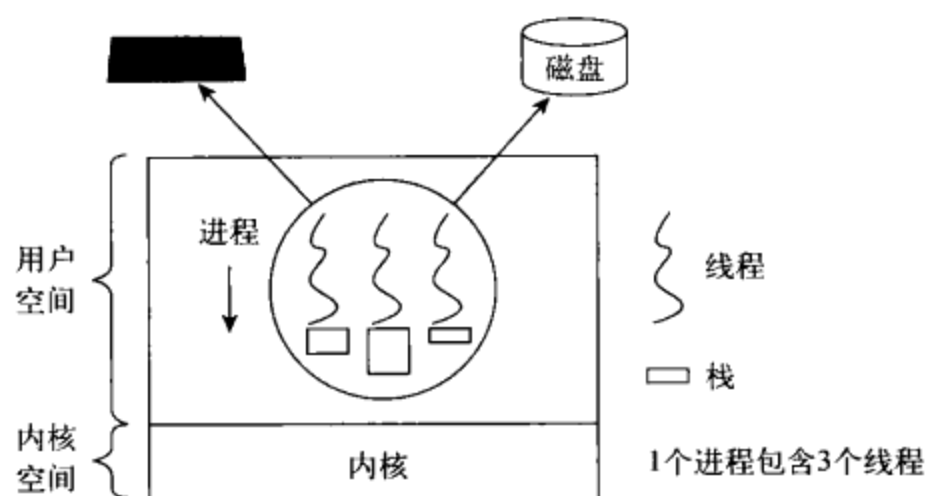


图 5-3 文本处理进程的三个线程

5.2 线程管理

有进程后,要管理进程。那么有线程后,也要进行管理。而管理的基础也与进程管理的基础类似:就是要维持线程的各种信息。这些信息包含了线程的各种关键资料。存放这些信息的数据结构称为线程控制表或线程控制块。那么线程控制块里面到底包含哪些信息呢?

我们说过线程共享一个进程空间,因此,许多资源是共享的。这些共享的资源显然不需要存放在线程控制块里面,而是存放在进程控制块即可。但由于线程是不同的执行序列,总会有些不能共享的资源。就像一家的兄弟姐妹。家里很多东西都是共享,如所有人同住父母的房子,共用冰箱、彩电、餐桌等。但有的东西则是每个人独享的,如衣服、日记本等。这些不被共享的资源和信息就需要存放在线程控制块里。

到底哪些资源可由(同一进程的)不同线程所共享,哪些不可共享呢?这当然是仁者见仁,智者见智。但也是有规律的。这个规律就是应当让共享的资源越多越好,因为这是我们发明线程的主要动机之一。由于我们发明线程的目的就是要经常协作,共享自然是我们的不懈追求。因此,一般的评判标准是:如果某资源不独享会导致线程运行错误,则该资源就由每个线程独享;而其他资源都由进程里面的所有线程共享。

按照这个标准来划分,线程共享的资源有地址空间、全局变量、文件、子进程等。定时器,信号和占用 CPU 时间也可以共享。但程序计数器不能共享,因为每个线程的执行序列不一样。同理,寄存器也不能共享,栈也不能共享,这是线程的上下文(运行环境)。表 5-1 给出的是一般情况下(同一进程的)线程间共享和独享资源的划分。

表 5-1 一般情况下线程共享和独享资源划分

线程共享资源	线程独享资源
地址空间	程序计数器
全局变量	寄存器
打开的文件	栈
子进程	状态字
闹铃	
信号及信号服务程序	
记账信息	

5.3 线程的实现方式

既然线程是进程的构成部分,或者是进程的分身,那么由谁来管理线程就有两种选择:一是让进程自己来管理线程;二是让操作系统来管理线程。这种不同的选择就出现了内核态线程和用户态线程。这也是线程实现的两种方式。由进程自己管理就是用户态线程实现,由操作系统管理就是内核态线程实现。

细心的读者也许已经注意到,我们在讲述进程时没有提到过实现方式的问题,即是应该在用户态还是内核态实现的问题。这是因为进程是在 CPU 上实现并发(多道编程),而 CPU 是由操作系统管理的,因此,进程的实现只能由操作系统内核来进行,而不存在用户态实现的情况,根本没有这种探讨的需要。但对于线程就不同了,因为线程是进程内部的东西,当然存在由进程直接管理线程的可能性。这就是为什么我们要探讨线程内核态与用户态实现。

5.3.1 内核态线程实现

前面说过,线程是进程的分身,是进程的不同执行序列。既然每个线程是不同的执行序列,说明线程应该是 CPU 调度的基本单位。我们知道,CPU 调度是由操作系统实现的。因此,让操作系统来管理线程似乎是天经地义的事情。

那么操作系统怎么管理线程呢?与管理进程一样,操作系统要管理线程,就要保持维护线程的各种资料,即将线程控制块存放在操作系统内核空间。这样,操作系统内核就同时保有进程控制块和线程控制块。而根据进程控制块和线程控制块提供的信息,操作系统就可以对线程进行各种类似进程的管理,如线程调度、线程的资源分配、各种安全措施的实现等。图 5-4 描述的就是内核态线程的实现示意图。

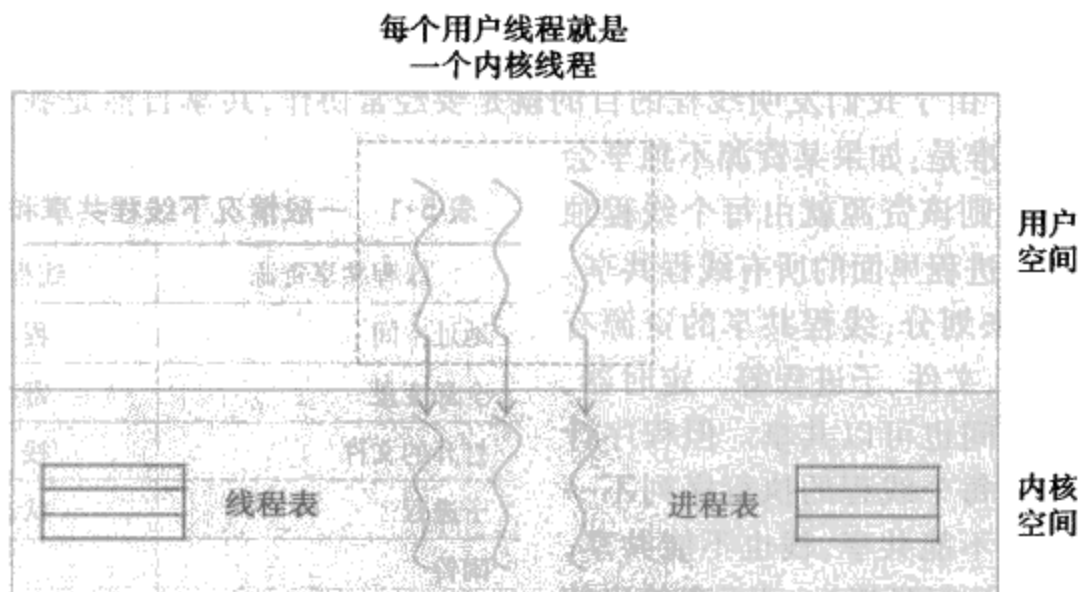


图 5-4 内核态线程实现

由操作系统来管理线程有很多好处,最重要的好处是用户编程保持简单。因为线程的复杂性由操作系统承担,用户程序员在编程时无需管理线程的调度,即无需担心线程什么时候会执

行、什么时候会挂起。另外一个重要好处是如果一个线程执行阻塞操作,操作系统可以从容地调度另外一个线程执行。因为操作系统能够监控所有的线程。

那么内核态线程实现有什么缺点呢?有。首先是效率较低。因为线程在内核态实现,每次线程切换都需要陷入到内核,由操作系统来进行调度。而从用户态陷入到内核态是要花时间的。另外,内核态实现占用内核稀缺的内存资源,因为操作系统需要维护线程表。操作系统所占内存空间一旦装载结束后就已经固定,无法动态改变。由于线程的数量通常大大高于进程的数量,那随着线程数量的增加,操作系统内核空间将迅速耗尽。

如果要建立进程线程,但内核空间不够了,怎么办?我们可以做的选择有:“杀死”别的进程;创建失败;让它等一下。前面说过,“杀死”别的进程是一件很不好的事情,因为将造成服务不确定性。宣称创建失败也很差。因为创建失败有可能意味着某个进程无法往前推进,这违反了我们前面说过的进程模型的时序推进要求。让创建者等一下,这要看创建的是什么进程和线程了。如果是系统进程线程,等一下可能意味着关键服务无法按时启动;如果是用户进程线程,等一下可能引起用户的强烈不满。而且,等多久谁也不知道。

那在内核空间满了后,应该怎么办呢?打一个战场上的比方就清楚了。如果战场上对手太厉害了,想再调个师,结果没有,怎么办?投降。也就是说,如果内核空间溢出,操作系统将停止运转。因为要创立的进程可能很重要,又不能不创建。所以最好的结局是“死掉”。别人发现系统死了就会采取行动来补救。如果操作系统还要运转,却不能正确地运转,那是很危险得事情。操作系统采取的这种行为在灾难应对领域称为“无害遽止”。

但上面两个缺点还不是最要命的。最要命的是内核态实现需要修改操作系统,这在线程概念提出之初是一件很难办到的事情。试想,如果你作为研究人员提出了线程概念,然后你去找一家操作系统研发商,要求其修改操作系统,加入线程的管理,结果会怎样?操作系统开发商会请你走开。有谁敢把一个还未经证明的新概念加入到对计算机影响甚大的操作系统里?除非我们先证明线程的有效性,否则很难说服他人修改操作系统。

这样,就有了线程的用户态实现。

5.3.2 用户态线程实现

线程在刚刚出现时,由于无法说服操作系统人员修改操作系统,其实现的方式只能是在用户态。(谁提出谁举证。)那么用户态实现意味着什么呢?或者说用户态实现是什么意思呢?就是用户自己做线程的切换,自己管理线程的信息,而操作系统无需知道线程的存在。

那么在用户态如何进行线程调度呢?那就是用户自己写一个执行系统(runtime system)作调度器(runtime scheduler),即除了正常执行任务的线程外,还有一个专门负责线程调度的线程。由于大家都在用户态下运行,谁也不比谁占优势,要想取得CPU控制权只能靠大家的自愿合作。一个线程在执行完一段时间后主动把资源释放给别人使用,而在内核态下则无需如此。因为操作系统可通过周期性的时钟中断把控制权夺过来。在用户态实现情况下,执行系统的调度器也是线程,没有能力强行夺走控制权。所以必须合作。图5-5描述的是用户态线程的实现示意。

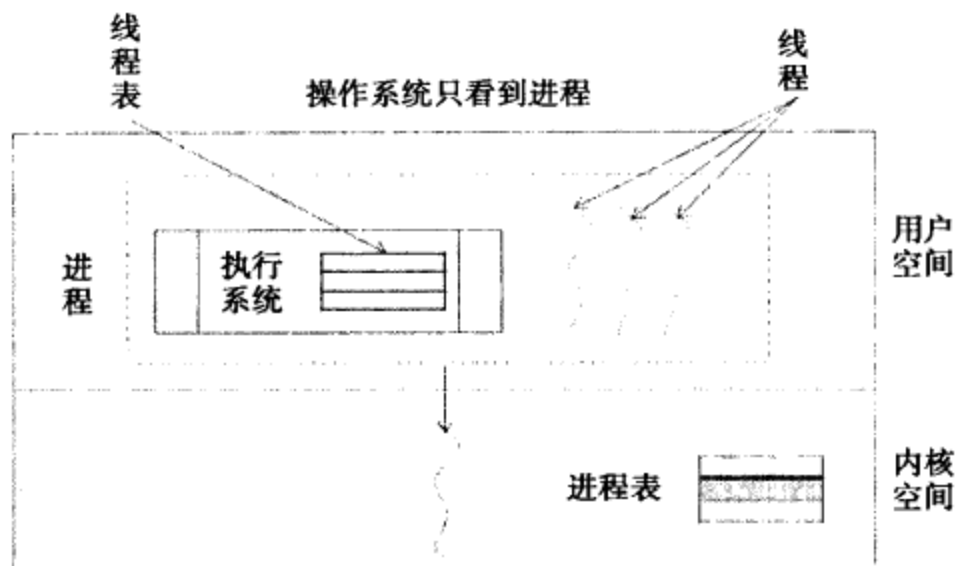


图 5-5 用户态线程实现

那么用户态实现有什么优点呢？有。首先是灵活性。因为操作系统不用知道线程的存在，所以在任何操作系统上都能应用；第二个优点是线程切换快，因为切换在用户态进行，无需陷入到内核态。第三，不用修改操作系统，实现容易。

那么这种实现方式有什么缺点吗？有。首先编程序变得很诡异。我们前面说过，用户态线程需要相互合作才能运转。这样，我们在写程序时，必须仔细斟酌在什么时候应该让出 CPU 给别的线程使用。而让出时机的选择对线程的效率和可靠性有很大的影响。这并不是件容易的事。另外一个更为严重的问题是，用户态线程实现无法完全达到线程提出所要达到的目的：进程级多道编程。

如果在执行过程中一个线程受阻，它将无法将控制权交出来（因为受阻后无法执行交出 CPU 的指令了），这样整个进程都无法推进。操作系统随即把 CPU 控制权交给另外一个进程。这样，一个线程受阻造成整个进程都受阻，我们期望的通过线程对进程实施分身的计划就失败了。这是用户态线程的致命弱点。

但是，作为线程的提出者，自然不愿意这么快就承认线程的概念破产。因此，总要想点办法来挽救。那有什么办法来挽救呢？既然线程阻塞造成整个进程阻塞，解决的办法只有两种：一是不让线程阻塞；二是阻塞后想办法激活同一进程的另外线程。

第一种办法如何实现呢？有几种办法。首先来看线程阻塞的原因。

线程之所以阻塞是因为它执行了阻塞操作，如读写磁盘、收发数据包等。那我们就想，如果将这些操作改为非阻塞操作，不就解决问题了吗。但是这种办法根本就行不通。首先，将所有系统调用改为非阻塞就得修改操作系统，而我们刚才说了，用户态线程实现就是不想修改操作系统；第二，就算操作系统的人员很仁慈，帮你修改，那可以吗？不可以，因为很多系统调用的语义里面就包括阻塞，即阻塞是其正确运行的前提。使用这些系统调用的程序期望着阻塞，而修改系统调用的语义就会造成这些程序运行错误。所以这个建议行不通。

既然不能将阻塞操作修改为非阻塞操作，那我们可以不让线程调用阻塞操作。我们只需要在线程进行任何系统调用前，先行确认一下该调用是否会发生阻塞，即我们写一个包裹（wrap），将系统调用包裹起来，用户程序使用系统调用时需通过这个 wrap。wrap 里有一段代码，专门检

查发出的系统调用会不会阻塞。如果会,就禁止调用;否则,就放行。

例如,在读写磁盘时,先行执行一个小的检查程序,看看需要的数据是否可以迅速获得(如检查一下排在前面的磁盘读操作有多少),如果是,就放行该操作。否则,先切换给别人,到不会阻塞了再调用。这样就可以一定程度地缓解线程阻塞造成进程阻塞的问题(但并没有完全解决,知道为什么吗?)。但这样做有很大的缺点:一是,需要修改操作系统,将系统调用 wrap 包裹起来。二是,这样做大大降低了线程的效率。三是,这种做法有个关键前提条件。因为我们不让程序发出阻塞调用并不是要永远不让该线程运行,而是让它等一段时间。因此,本做法隐含的前提条件是你等一段时间后该调用就会由阻塞调用变成非阻塞调用,否则的话,该程序就永远不能运转了。而这个前提假定却不一定成立。

当然了,有的调用在等待一段时间后再调用确实会变成非阻塞操作。例如,你想从网络上接受一个数据包,但是发送方尚未发送,这个时候你如果使用 receive 系统调用(假如我们使用阻塞版的 receive)将发生阻塞。但如果过一段时间后你再调用,这个包可能已经发出来了,你的调用就是非阻塞了。但问题是,并不是所有的阻塞调用在等待一段时间后都会转变成非阻塞。比如读磁盘,你一调用就阻塞,但如果你不调用,就不会读磁盘,那么你在将来任何时候读磁盘仍将阻塞,这样该线程就永远无法推进。

既然不让线程阻塞的两种办法都不怎么样。那就来分析第二种解决办法:即在进程阻塞后想办法激活受阻进程的其他线程。这种办法的实现必须依赖操作系统。因为线程阻塞后,CPU 控制权已经回到操作系统手里。而要激活受阻进程的其他线程,唯一的办法是让操作系统在进行进程切换时先不切换,而是通知受阻的进程执行系统(即调用执行系统),并问其是否还有别的线程可以执行。如果有,将 CPU 控制权交给该受阻进程的调用执行系统,从而调度另一个可以执行的线程到 CPU 上。这种做法称为调度器激活(Scheduler Activation),因为我们所干的事情就是激活进程里面的调度器(调用执行系统)。

我们将这种做法称为“第二机会”。因为在一个进程挂起后,操作系统并不立即切换到别的进程,而是给该进程第二次机会,让其继续执行。如果该进程只有一个线程,或者其所有线程都已经阻塞,则控制权将再次返回给操作系统。而这次,操作系统就会切换到别的进程了。

这种办法似乎解决了阻塞线程阻塞进程的问题。但也有两个缺点:首先还是需要修改操作系统,使得其在进行进程切换时,不是立即切换到别的进程,而是调用受阻进程的调用执行系统。但由于此种修改范围小,只需要对调度器程序做一个外科手术式的小改动,因而尚可以忍受。

但该做法还存在一个更为严重的缺陷:这种操作系统调用用户态调用执行系统的做法违反了我们所遵循的层次架构原则。因为这种调用属于所谓的 up-call,即下层功能调用了上层功能(操作系统在下,调用执行系统在上)。而平时用户程序使用操作系统服务的调用属于 down-call,即上层程序调用下层服务。这种违反上下有别的做法使得操作系统的设计和管理都变得复杂,而且,由于调度器在第一次切换时总是选择阻塞的进程,这样也为黑客和各种攻击者提供了一个系统缺口。另外,这种层次结构的违反让习惯了上下有别的人类感到十分不快,因此,此种做法没有得到商用操作系统的认可。

5.4 现代操作系统的线程实现模型

鉴于用户态和内核态都存在缺陷,现代操作系统使用的是将二者结合起来。用户态的执行系统负责进程内部线程在非阻塞时的切换;内核态的操作系统负责阻塞线程的切换,即我们同时实现内核态和用户态线程管理。其中内核态线程数量较少,而用户态线程数量多。每个内核态线程可以服务一个或多个用户态线程。换句话说,用户态线程被多路复用到内核态线程上。例如,某个进程有5个线程,我们可以将5个线程分成两组,一组3个线程,另一组2个线程。每一组线程使用一个内核线程。这样,该进程将使用两个内核线程。如果一个线程阻塞,则与其同属于一组的线程皆阻塞,但另外一组线程却可以继续执行,如图5-6所示。

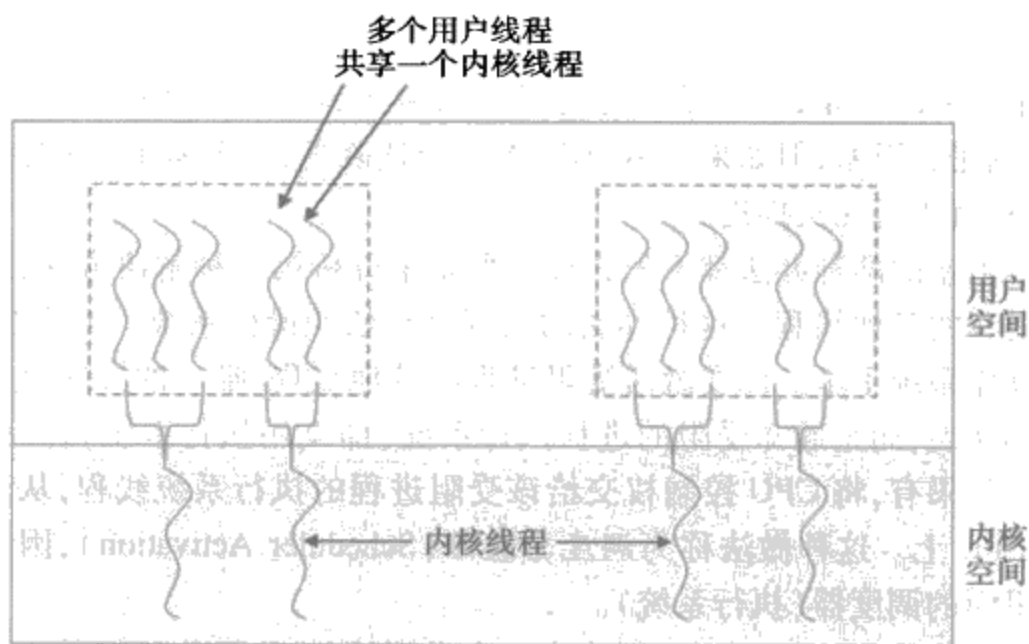


图 5-6 线程的内核态与用户态混合实现

这样,在分配线程时,我们可将需要执行阻塞操作的线程设为内核态线程,而不会执行阻塞操作的线程设为用户态线程。这样我们就可以获得两种态势实现下的优点,而避免其缺点。

5.5 多线程的关系

推出线程模型的目的就是为了实现进程级并发。因为在一个进程中通常会出现多个线程,否则,我们也没有必要研究什么线程了。就像看舞台剧,如果只跳上来一个人,从头演到尾,没有其他人上场,大部分观众都会觉得无聊。我们要看的是演员对不同人物的刻画,色彩斑斓。因此,研究线程就要研究多线程,多个线程共享一个舞台,时而交互、时而独舞。

但共享一个舞台会带来不必要的麻烦。就像人们共享资源时难免产生争端一样。线程在共享地址空间的过程中也会产生矛盾。这些矛盾归结为下面两个根本问题:

- 线程之间如何通信?
- 线程之间如何同步?

而上述两个问题在进程层面也同样存在。从一个更高的层次上看,不同的进程也共享着一个巨大的空间,这个空间就是整个计算机。因此,进程之间也会存在矛盾,而这些矛盾也体现在如何通信(沟通)和如何同步(协调)上。

本书接下来的两章就分别对进程线程的通信和同步进行详细论述。

5.6 讨论:从用户态进入内核态

什么情况会造成一个线程从用户态进入到内核态呢?

首先,如果在程序运行过程中发生中断或异常,系统将自动切换到内核态来运行中断或异常处理机制。图 5-7 描述的就是中断导致态势切换的流程。异常处理的流程与此相同或相似。

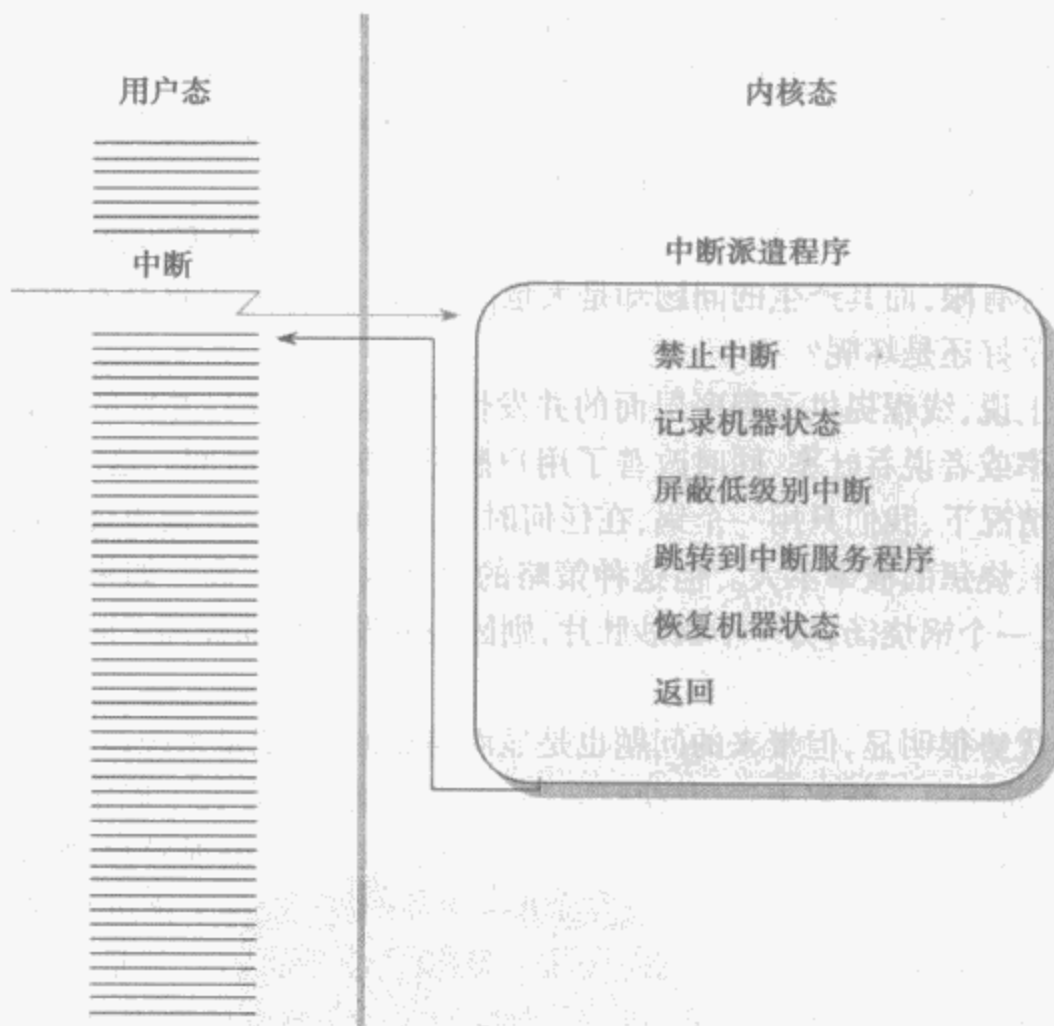


图 5-7 中断导致程序运行从用户态切换到内核态

此外,程序进行系统调用也将造成从用户态进入到内核态的转换。例如,一个 C++ 程序调用函数 `cin`。`cin` 是一个标准库函数,它将调用 `read` 函数。而 `read` 则是由操作系统提供的一个系统调用。其执行过程如下:

- 1) 执行汇编语言里面的系统调用指令(如 `syscall`)。
- 2) 将调用的参数 `sys_read`, `file number`, `size` 存放在指定的寄存器或栈上(事先约好)。
- 3) 当处理器执行到“`syscall`”指令时,察觉这是一个系统调用指令,将进行如下操作:

- a) 设置处理器至内核态。
 - b) 保存当前寄存器(栈指针、程序计数器、通用寄存器)。
 - c) 将栈指针设置指向内核栈地址。
 - d) 将程序计数器设置为一个事先约定的地址上。该地址上存放的是系统调用处理程序的起始地址。
- 4) 系统调用处理程序执行系统调用,并调用内核里面的 read 函数。
- 这样,就实现了从用户态到内核态的转换,并完成系统调用所要求的功能。

5.7 讨论:线程的困惑——确定性与非确定性

讲完这一章后,读者应该得到一个结论:线程是很有用的东西,因为实现了进程内部的并发。但这个结论是真的吗?线程让进程有了分身术,它在进程级别上实现了多道编程,使得一个进程可以同时干多件事情,提高了程序运行的效率,提高了硬件资源的使用率。似乎,线程带给我们的只有优点,或者说线程带给我们的优点远远大于其缺点。

但真的是这样吗?有人不这样认为。2006 年夏天,有人在 IEEE Computer 上发表文章,说线程带给我们的优势有限,而其产生的问题却是大量的,应该剔除掉,说这是万恶之源。

那线程到底是好还是坏呢?

从某种程度上说,线程提供了程序层面的并发性能。毫无疑问,并发的好处是显而易见的,既提高系统的效率或者说吞吐率,同时改善了用户感觉到的响应时间。这有点像一个人烧菜的情况。在单线程情况下,我们只用一个锅,在任何时刻,只有一个菜在锅里烧,我们可以专注于这一个菜,发生烧糊、烧焦的概率不大。但这种策略的时间效率也不高。如果我们同时用几个锅,一个锅烧红烧肉,一个锅烧汤,另一个锅炒肚片,则因为并发操作而提高了时间效率,即可以同时烧好三个菜。

虽然线程的优势很明显,但带来的问题也是显而易见的,那就是系统运行的不确定性。由于多线程的存在,就每个单一线程来看,其执行效率、执行正确性均存在不确定性。当然,通过使用同步机制,可以改善这种不确定性。但如果在多线程执行过程中出现异常,则情况就相当麻烦。另外,在多线程下,如果某个进程的某个线程创建一个子进程,那对于该进程的其他线程来讲意味着什么?这是一个众说纷纭,莫衷一是的话题。用烧菜的例子来说,就是一般情况下我们也许能够掌控局势,使得每个菜都烧得不错。但问题是,如果因为环境的原因,如火候、水量等原因,造成红烧肉干底,汤溢出,则将造成手脚忙乱,顾此失彼,从而导致烧菜失败。

如果读者对计算机硬件与体系结构熟悉,可能能够看出来,线程的机制非常类似于硬件的流水线机制。流水线也是提供并发,不过是指令级的(而不是程序级的)。并发当然提高了计算机的吞吐率,改善了用户响应时间。但问题是,在多流水线多梯级情况下,由于有许多指令同时不同的流水线和梯级上执行,其之间存在的数据和指令依赖关系十分复杂。如果万一再发生异常,如何保存一个一致性的状态都成了问题。图 5-8 给出的是流水线的指令级并发和线程的程序级并发的对照。

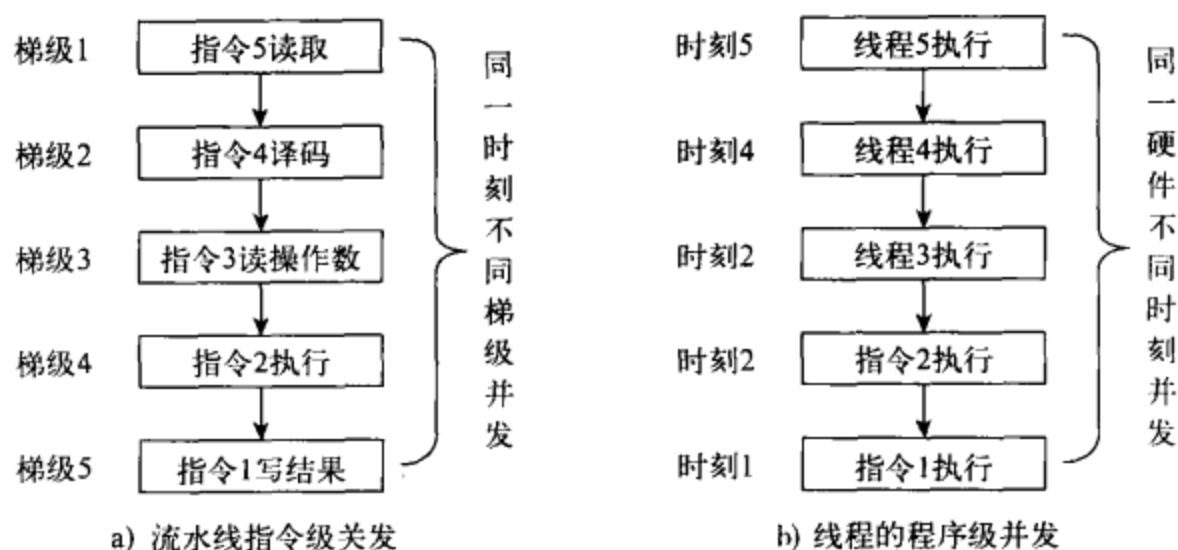


图 5-8 线程与流水线的相似性

更为重要的是,线程和流水线的管理十分复杂。读者已经看到,线程的同步机制繁多、复杂,使得整个操作系统都变得更为复杂,从而增加整个系统的不可靠性。在硬件层,流水线也一样,需要许多复杂技术的支持,如多指令发射里的超长指令字与超标量计算等,这使得编译器或指令级结构或者两者同时变得复杂。复杂的东西其可靠性是很难保证的。

从某种程度上说,线程与流水线分别是软件层和硬件层不确定性的根源。流水线使得我们可以在硬件指令执行上并发,线程则使我们在软件指令执行上并发。但其带来的操作系统、编译系统和指令集结构的高度复杂是否值得,也许并不容易回答。

思考题

1. 线程与进程的相似和不同之处何在? 请予以详细论述。
2. 用户态线程实现由于不能应对阻塞,有人认为是没有任何用处的。你持何种看法?
3. 调度器激活(Scheduler Activation)是一种良好的解决用户态线程问题的方案吗? 它为什么没有得到商业应用?
4. 一个进程里面的不同线程是否受到相互保护? 为什么?
5. 两个不同进程里面的线程是否受到相互保护? 为什么?
6. 一个进程中的不同线程如何共享信息? 这种共享可以在不同进程的不同线程之间直接实现吗?
7. 如果将整个计算机看作是一个巨大的进程,则运行在上面的正常进程就相当于这个巨大进程里面的子进程。请比较此种进程观的优缺点。
8. 从某种程度上,人类社会里的一个家庭可以看做是一个进程,而家庭成员则是线程。请根据对家庭的体验论述线程之间资源共享有可能存在的矛盾。
9. 请解释 wrap 在线程实现上的作用,其实用价值大吗?
10. 在创建线程时,如果内核空间已满,我们可以等待一段时间后再试图创建。请讨论这种策略的优缺点及可行性。
11. 请调查几种商用操作系统的线程实现模型,并予以论述。
12. 分析:当一个线程从用户态进入到内核态时,需要进行哪些切换?

第6章 线程通信

引子

1995年4月24日,纽约时报和华盛顿邮报非常不情愿地,同时刊登了一封长度达4个版面的文章,标题是“工业社会及其未来”。文章作者是西奥多约翰·卡扎斯基,一个被美国联邦调查局搜捕了17年之久的“孤独的爆破手”(见图6-1)。该作者在从1978至1995年的17年间,通过发送炸弹邮件,炸死炸伤几十名学者、科技公司高管和技术倡导人士。

作为哈佛大学学士、密歇根大学博士,卡扎斯基于25岁成为伯克利加州大学(UC-Berkeley)数学教授。“无论说他怎么聪明都不过分”,这是熟悉他的教授、同事给出的评价。就是这样一个人,却在1969年突然从UC-Berkeley消失,并随后隐居美国蒙大拿州乡间。他在使用其他方式反对技术对人类的威胁都遭到失败后,于1978年开始通过自制炸弹来袭击技术界人士,以表达他反对技术带给人类各种灾难的立场。而联邦调查局也随即开始了该局历史上耗时最长、花费最为昂贵的搜捕行动。但却一无所获,孤独的爆破手的炸弹仍然继续发挥着威力。



图6-1 孤独的爆破手卡扎斯基
(Theodore Kaczynski)

而卡扎斯基的文章在纽约时报发表后,他的弟弟认出了其笔迹和信仰,报告了联邦调查局,卡扎斯基从而被捕。联邦调查局历史上最长最昂贵的搜捕行动宣告结束。

卡扎斯基因为其沟通的欲望而落入法网,可见沟通对人的重要性。而进程和线程也不例外,它们对沟通的需求也非常强烈。

6.1 为什么要通信

通信是人的基本需求。古罗马皇帝的实验证明,没有与人交互的婴儿无法存活。美国“孤

独的爆破手”卡扎斯基就因为其强烈的与人沟通的欲望,迫使纽约时报发表了其立场宣言,从而导致被联邦调查局捕获。

而进程作为人的发明,自然脱离不了人的习性,也有通信需求。如果进程之间不进行任何通信,那么进程所能完成的任务就要大打折扣。例如,父进程在创建子进程后,通常须要监督子进程的状态,以便在子进程没有完成给定的任务时,可以再创建一个子进程来继续。这就需要父子进程间通信。

而线程间的通信则需要更多。由于一个进程通常包括多个线程,这多个线程之间因资源共享自然地就存在一种合作关系。这种合作关系虽然可以表现为相互独立,但更多地时候是互相交互。这就是通信。就像舞台上的多个演员,他们之间是一种合作关系,共同将戏演好。虽然这些演员在舞台上的时候可以各自演各自的,不说话,也没有肢体接触,即没有交互,但他们更多的时候会进行对白和拥抱等交互操作。

线程之间的交互我们就称之为线程通信。线程通信是从进程通信演变而来的,进程通信有个专有缩写,叫IPC(Inter-Process Communication)。由于每个进程至少有一个线程,进程的通信就是进程里面的线程通信。在随后的讨论中,我们将统一使用线程通信来进行讲解。

那么线程之间的通信是如何进行的呢?

舞台上的演员可以通过对白、手势和拥抱等方法来交互通信。类似地,线程也可以同样的方式来进行通信。下面我们就来看一下线程的这些交互方式。

6.2 线程对白:管道、记名管道、套接字

演员最常使用的交互手段就是对白。对白就是一方发出声音,另一方接受声音。声音的传递则通过空气(当面或无线交谈)、线缆(有线电话)进行传递。类似地,线程对白就是一个线程发出某种数据信息,另外一方接受数据信息,这些数据信息通过一片共享的存储空间进行传递。

在这种方式下,一个线程向这片存储空间的一端写入信息,另一个线程从存储空间的另外一端读取信息。这看上去像什么?管道。管道所占的空间既可以是内存,也可以是磁盘(见图6-2)。就像两人对白的媒介可以是空气,也可以是线缆一样。要创建一个管道,一个线程只需调用管道创建的系统调用即可。



图6-2 操作系统里的管道类似生活中的管道

6.2.1 管道

从根本上说,管道是一个线性字节数组,类似文件,使用文件读写的方式进行访问。但却不是文件。因为通过文件系统看不到管道的存在。另外,我们前面说了,管道可以设在内存里,而

文件很少设在内存里(当然,有研究人员在研发基于内存的文件系统,但这个还不是主流)。

创建管道在壳命令行下和在程序里是不同的。壳命令行下,只需要使用符号“|”即可。例如,在 UNIX 壳下,我们可以键入如下命令:

```
$ sort <file1 |grep zou
```

在两个 utility “排序(sort)”和“查找(grep)”之间创建了一个管道,数据从 sort 流向 grep。sort 的结果将作为 grep 的输入。上述命令的意思是对 file1 的内容进行排序,排序完的结果作为 utility 程序 grep 的输入,在结果里面找出所有包括字符串 zou 的文本行。

在程序里面,创建管道需要使用系统调用 popen() 或者 pipe()。popen 需要提供一个目标进程作为参数,然后在调用该函数的进程和给出的目标进程之间创建一个管道。这很像人们打电话时必须提供对方的号码,才能创建连接一样。

创建时还需要提供一个参数表明管道类型:读管道或者是写管道。而 pipe 调用将返回两个文件描述符(文件描述符是用来识别一个文件流的一个整数,与句柄不同),其中一个用于从管道进行读操作,一个用于写入管道。也就是说,pipe 将两个文件描述符连接起来,使得一端可以读,另一端可以写。通常情况下,在使用 pipe 调用创建管道后,再使用 fork 产生两个进程,这两个进程使用 pipe 返回的两个文件描述符进行通信。

例如,下述代码段创建一个管道并利用它在父子进程间通信。

```
int pp[2];
pipe(pp);                /* 创建管道 */
if(fork() == 0){          /* 子进程 */
    read(pp[0]);           /* 从父进程读 */
    .....
}
else{
    write(pp[1]);          /* 写给子进程 */
    .....
}
```

管道的一个重要特点是使用管道的两个线程之间必须存在某种关系,例如,使用 popen 需要提供另一端进程的文件名,使用 pipe 的两个线程则分别隶属于父子进程。

6.2.2 记名管道

如果要在两个不相关的线程,如两个不同进程里面的线程,之间进行管道通信,则需要使用记名管道。顾名思义,记名管道是一个有名字的通信管道。记名管道与文件系统共享一个名字空间,即我们可以从文件系统中看到记名管道。也就是说,记名管道的名字不能与文件系统里的任何文件名重名。例如,在 UNIX 下使用 ls 命令可以查看到已经创立的记名管道。

```
% ls-l fifol
prw-r--r-- 1 john users 0 Sep 22 23:11 fifol
```

一个线程通过创建一个记名管道后,另外一个线程可使用 open 来打开这个管道(无名管道则不能使用 open 操作),从而与另外一端进行交流。

记名管道的名称由两部分组成:计算机名和管道名,例如\\[主机名]\\管道\\[管道名]。对于同一主机来讲允许有多个同一命名管道的实例并且可以由不同的进程打开,但是不同的管道都有属于自己的管道缓冲区而且有自己的通信环境,互不影响。命名管道可以支持多个客户端连接一个服务器端。命名管道客户端不但可以与本机上的服务器通信也可以同其他主机上的服务器通信。

管道和记名管道虽然具有简单,无需特殊设计(指应用程序方面)就可以和另外一个进程进行通信的优点,但其缺点也是显然的。首先是管道和记名管道并不是所有操作系统都支持。主要支持管道通信方式的是 UNIX 和类 UNIX(如 Linux)的操作系统。这样,如果需要在其他操作系统上进行通信,管道机制就多半会力不从心了。其次,管道通信需要在相关的进程间进行(无名管道),或者需要知道按名字来打开(记名管道),而这在某些时候会十分不便。

6.2.3 虫洞:套接字

套接字(socket)是另外一种可以用于进程间通信的机制。套接字首先在 BSD 中出现,随后几乎渗透到所有主流操作系统。套接字的功能非常强大,可以支持不同层面、不同应用、跨网络的通信。使用套接字进行通信需要双方均创建一个套接字,其中一方作为服务器方,另外一方作为客户方。服务器方必须先创建一个服务区套接字,然后在该套接字上进行监听,等待远方的连接请求。欲与服务器通信的客户则创建一个客户套接字,然后向服务区套接字发送连接请求。服务器套接字在收到连接请求后,将在服务器机器上创建一个客户套接字,与远方的客户机上的客户套接字形成点到点的通信通道。之后,客户端和服务端就可以通过 send 和 recv 命令在这个创建的套接字通道上进行交流了。

服务区套接字有点类似于传说中的虫洞(worm hole),如图 6-3 所示。虫洞的一端是开放的,它在宇宙内或宇宙间飘移着,另外一端处于一个不同的宇宙,监听是否有任何东西从虫洞来。而欲使用虫洞者需要找到虫洞的开口端(发送连接请求),然后穿越虫洞即可。

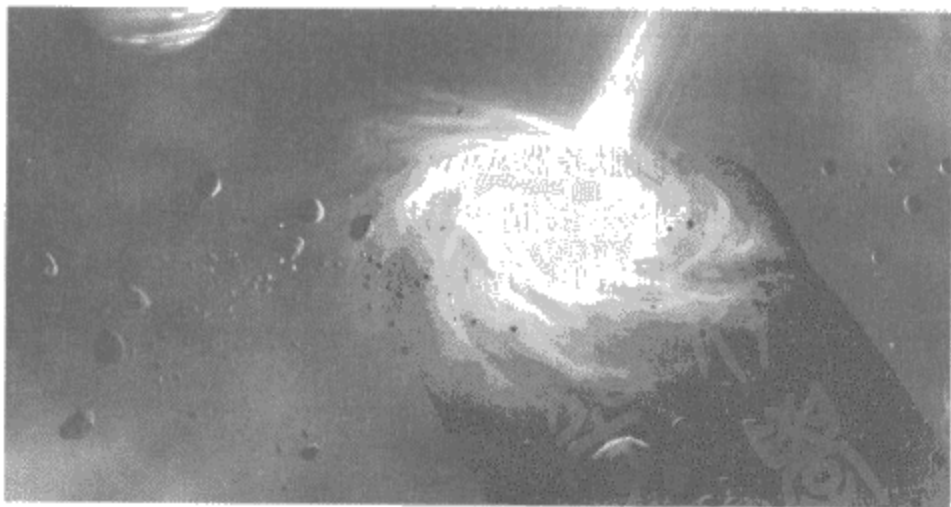


图 6-3 服务器套接字与宇宙虫洞类似

使用套接字进行通信稍微有点复杂,我们下面以一个网页浏览的例子对套接字这种通信方式予以说明。对于一个网站来说,要想提供正常的网页浏览服务,其网站服务器需要首先创建一个服务器套接字,作为外界与本服务器的通信信道。为了使该信道为外人所知,我们通常将该服

务器套接字与某公共主机的一个众所周知的端口进行绑定,如图 6-4 所示。

```
#创建一个名为 INET 的流套接字
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#将套接字与某公共主机的一个众所周知的端口(这里端口号为 80)绑定。
serversocket.bind((socket.gethostname(), 80))
```

图 6-4 网站服务器创建服务器套接字并与端口 80 进行绑定

图 6-4 进行套接字和端口绑定的语句里的 `socket.gethostname()` 用来将套接字向外界公开。如果将该语句的使用 `socket.gethostname()` 改为 `'` 或者 `localhost` 或者某个具体的 IP 地址如 `(120.121.4.1)`, 则该服务器套接字将被限制在本地机器上使用。

在创建了服务器套接字并将其向外界打开后,网站服务器即可以在该套接字上进行监听,如图 6-5 所示。

```
serversocket.listen(5) /* 将套接字变为一个服务区套接字 */
```

图 6-5 在打开的套接字上进行监听

图中语句里的数字 5 将端口上的等待队列长度限制为 5, 即超过 5 个的请求将被拒绝。

到这里,服务器端的设置就宣告结束。

对于客户来说,如要访问该上述网站,需要点击该网站的网址。在点击网址后(我们这里假定该网站网址为 `www.sjtu.edu.cn`),客户机上的网络浏览器进行若干步操作,如图 6-6 所示。

```
/* 创建一个名为 INET 的流套接字 */
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
/* 链接到端口号为 80 的网站服务器端口(通常使用的 http 端口) */
s.connect(("www.sjtu.edu.cn", 80))
```

图 6-6 客户端网络浏览器创建客户机套接字并与网站连接

图 6-6 中的 `s.connect` 命令将向服务器 `www.sjtu.edu.cn` 在端口 80 打开的服务器套接字发送连接请求。而服务器端在接收到该连接请求后,将生成一个新的客户端套接字与该客户端套接字对接,从而建立一个套接字通信信道。图 6-7 为网站服务器上运行的主循环。

```
while True:
    (clientsocket, address) = serversocket.accept() /* 接受外部连接请求 */
    /* 对 clientsocket 进行相关操作,例如创建一个新线程来处理客户请求 */
    newct = client_thread(clientsocket)
    newct.run()
```

图 6-7 网站服务器上运行的主循环

至此,套接字通信信道成功建立。客户端程序可以使用套接字 `s` 发送请求,索取网页,而服务器端则使用套接字 `clientsocket` 进行发送和接受消息。

这里需要指出的是服务器套接字既不发送数据,也不接收数据(指不接受正常的用户数据,而不是连接请求数据),而仅仅是生产出“客户”套接字。当其他(远方)的客户套接字发出一个连接请求时,我们就创建一个客户套接字。一旦客户套接字 `clientsocket` 创建成功,与客户的通信任务就交给了这个刚刚创建的客户套接字。而原本的服务器套接字 `serversocket` 则回到其原来的监听操作上。

套接字由于其功能强大而获得了很大发展,并出现了许多种类。不同的操作系统均支持或实现了某种套接字功能。例如按照传输媒介是否为本地,套接字可以分为本地(UNIX 域)套接字和网域套接字。而网域套接字又按照其提供的数据传输特性分为几个大类,分别是:

- 数据流套接字(`stream socket`):提供双向,有序、可靠、非重复数据通信。
- 电报流套接字(`datagram socket`):提供双向消息流。数据不一定按序到达。
- 序列包套接字(`sequential packet`):提供双向、有序、可靠连接,包有最大限制。
- 裸套接字(`raw socket`):提供对下层通信协议的访问。

套接字从某种程度上来说非常繁杂,各种操作系统对其处理并不完全一样。因此,如要了解某个特定套接字实现,读者需要查阅关于该套接字实现的具体手册或相关文档。

6.3 线程电报:信号

管道和套接字虽然提供了丰富的通信语义,并且也得到了广泛应用,但它们也存在某些缺点,并且在某些时候,这两种通信机制会显得很不好使。

首先,如果使用管道和套接字方式来通信,必须事先在通信的进程间建立连接(创建管道或套接字),这需要消耗系统资源。其次,通信是自愿的。即一方虽然可以随意往管道或套接字发送信息,对方却可以选择接收的时机。即使对方对此充耳不闻,你也奈何不得。再次,由于建立连接消耗时间,一旦建立,我们就想进行尽可能多的通信。而如果通信的信息量微小,如我们只是想通知一个进程某件事情的发生,则用管道和套接字就有点“杀鸡用牛刀”的味道,效率十分低下。

因此,我们需要一种不同的机制来处理如下通信需求:

- 想迫使一方对我们的通信立即作出回应。
- 我们不愿事先建立任何连接,而是临时突然觉得需要向某个进程通信。
- 传输的信息量微小,使用管道或套接字不划算。

应付上述需求,我们使用的是信号(`signal`)。

那么信号是什么呢?在计算机里,信号就是一个内核对象,或者说一个内核数据结构。发送方将该数据结构的内容填好,并指明该信号的目标进程后,发出特定的软件中断。操作系统接受到特定的中断请求后,知道是有进程要发送信号,于是到特定的内核数据结构里查找信号接受方,并进行通知。接到通知的进程则对信号进行相应处理。

信号非常类似我们生活当中的电报,如图 6-8 所示。如果你想给某人一封电报,你就拟好电文,将报文和收报人的信息都交给电报公司。电报公司则将电报发送到收报人所在地的邮局(中断),并通知收报人来取电报。发报时无需收报人事先知道,更无需进行任何协调。如果对方选择不对信号作出反应,则将终止操作系统运行。



图 6-8 信号非常类似人类生活中的电报派发与接收

6.4 线程旗语:信号量

信号量(Semaphore)是由荷兰人 E. W. Dijkstra 在 60 年代所构思出的一种程序设计构造。其原型来源于铁路的运行(见图 6-9):在一条单轨铁路上,任何时候只能有一列列车行驶在上面。而管理这条铁路的系统就是信号量。任何一列火车必须等到表明铁路可以行驶的信号后才能进入轨道。当一列火车进入单轨运行后,需要将信号改为禁止进入,从而防止别的火车同时进入轨道。而当火车驶出单轨后,则需要将信号变回到允许进入状态。这很像以前的旗语,如图 6-10 所示。

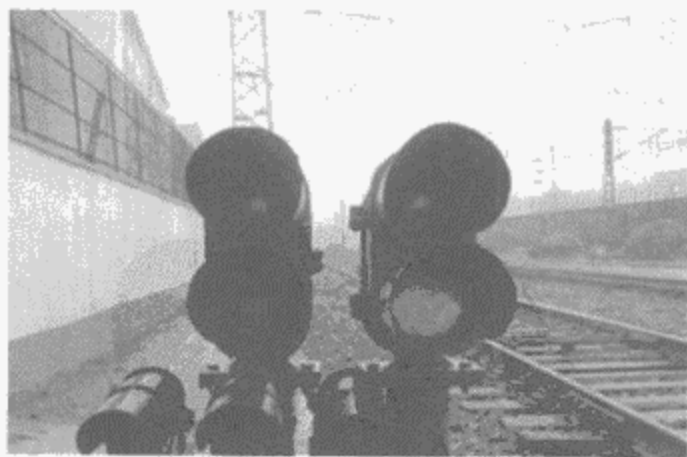


图 6-9 信号量来源于铁路信号系统

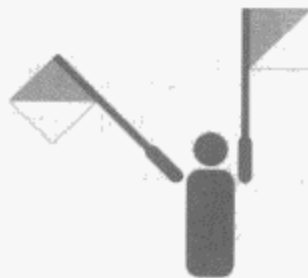


图 6-10 信号量相当于人类旗语

在计算机里,信号量实际上就是一个简单整数。一个进程在信号变为 0 或者 1 的情况下推进,并且将信号变为 1 或 0 来防止别的进程推进。当进程完成任务后,则将信号再改变为 0 或 1,从而允许其他进程执行。

需要提醒读者注意的是,信号量不光是一种通信机制,更是一种同步机制。本书在下一章进程同步时将再次论述信号量。

6.5 线程拥抱:共享内存

管道、套接字、信号、信号量,虽然满足了多种通信需要,但还是有一种需要未能满足。这就是两个进程需要共享大量数据。这就像两个人,他们互相喜欢,并想要一起生活时(共享大量数据),打电话、握手、对白等就显得不够了,这个时候需要的是拥抱,只有将其紧紧拥抱于怀,感觉才最到位,也才能尽可能地共享。

进程的拥抱就是共享内存(见图6-11)。共享内存就是两个进程共同拥有同一片内存。这片内存中的任何内容,二者均可以访问。要使用共享内存进行通信,一个进程首先创建一片内存空间专门作为通信用,而其他进程则将该片内存映射到自己的(虚拟)地址空间。这样,读写自己地址空间中对应共享内存的区域时,就是在和其他进程进行通信。

乍一看,共享内存有点像管道,有些管道不也是一片共享内存吗?这是形似而神不似。首先,使用共享内存机制通信的两个进程必须在同一台物理机器上;其次,共享内存的访问方式是随机的,而不是只能从一端写,另一端读,因此其灵活性比管道和套接字大很多,能够传递的信息也复杂得多。

共享内存的缺点是管理复杂,且两个进程必须在同一台物理机器上才能使用这种通信方式。共享内存的另外一个缺点是安全性脆弱。因为两个进程存在一片共享的内存,如果一个进程染有病毒,很容易就会传给另外一个进程。就像两个紧密接触的人,一个人的病毒是很容易传给另外一个人的。

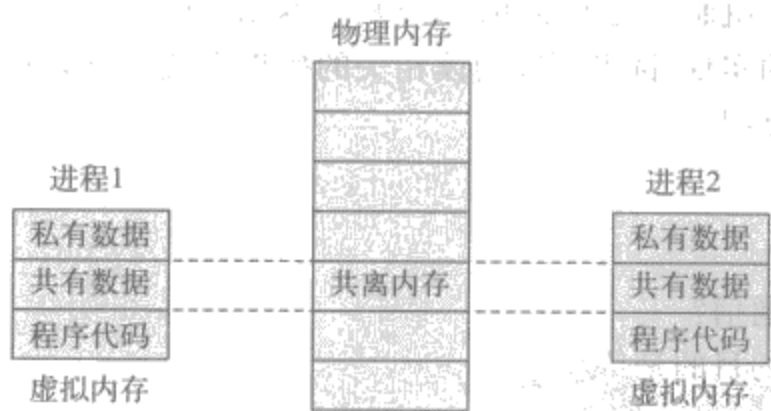


图6-11 共享内存

这里需要提请读者注意的是,使用全局变量在同一个进程的线程间实现通信不称为共享内存。

6.6 信件发送:消息队列

消息队列是一列具有头和尾的消息排列。新来的消息放在队列尾部,而读取消息则从队列头部开始,如图6-12所示。



图 6-12 消息队列

乍一看,这不是管道吗?一头儿读、一头儿写?没错。这的确看上去像管道。但它不是管道。首先它无需固定的读写进程,任何进程都可以读写(当然是有权限的进程)。其次,它可以同时支持多个进程,多个进程可以读写消息队列。即所谓的多对多,而不是管道的点对点。另外,消息队列只在内存中实现。

最后,它并不是只在 UNIX 和类 UNIX 操作系统实现。几乎所有主流操作系统都支持消息队列。

6.7 其他通信机制

除了上面介绍的主流通信方式外,有些操作系统还提供了一些其操作系统所特有的通信机制,例如 Windows 支持的进程通信方式就有剪贴板 (clipboard)、COM/DCOM、动态数据交换 (DDE)、邮箱 (mailslots);而 solaris 则有所谓的 solaris 门机制,让客户通过轻量级 (16KB) 系统调用使用服务器的服务。

虽然进程之间的通信机制繁多,且每种机制有着自己独特的特性,但归根结底都来源于 AT&T 的 UNIX V 系统。该系统在 1983 年加入了对共享内存、信号量和消息队列的支持。这三者就是众所周知的 System V IPC (POSIX IPC 也是源于该系统并成为当前 IPC 的标准)。因此,虽然不同操作系统的 IPC 机制可能不尽相同,但其基本原理则并无大的不同。如果需要了解具体操作系统的 IPC 机制的实现,读者可以阅读相关的操作系统内核教程。而本书则既不可能也无必要将它们全部进行介绍。

思考题

1. 进程为什么需要通信?
2. 线程通信要达到的目的是什么?
3. 消息队列和管道有何异同?
4. 管道的物理实体是什么?
5. 套接字和管道的异同点是什么? 可以用管道实现套接字的功能吗?
6. 请调查一个实际商用操作系统的套接字实现,并予以讨论。
7. 在本章介绍的所有通信机制里面,哪一种机制支持的信息交换量最大?
8. 试将本章介绍的通信机制按照出现的逻辑顺序排列出来,并简要说明其逻辑关系。
9. 有同学认为本章介绍的信号和信号量实际上是一回事,你同意吗? 为什么?
10. 请举出一个需要进程通信的编程实例。

第7章 进程同步

引子

1815年6月18日,比利时小城滑铁卢(见图7-1)。

法国皇帝拿破仑发出向由英国元帅惠灵顿指挥的第七联盟军进攻的命令。一时间,法军如潮水般涌向英军阵线,英军左右抵挡,勉力支持,形式十分危急。就在惠灵顿元帅准备放弃抵抗时,由布鲁切率领的普鲁士(今德国的一部)军队赶到,并加入战斗。在普鲁士有生力量的帮助下,英国军队振作起来,与普军联手击败了法军,从而彻底结束了拿破仑的政治生命。

普鲁士军队的姗姗来迟差点造成了英军的崩溃,同时,也由于普鲁士军队的最后加入拯救了由英国、俄国、瑞典、荷兰等14个国家和地区组成的第七联盟……

在战场上,友军之间的同步对战争的胜利至关重要。而在计算机里,线程之间的同步则对程序的正确运行至关重要。



图7-1 1815年6月18日的滑铁卢大战

7.1 为什么要同步

人类活动中不乏各种同步的需要。正是普鲁士军队与英国军队的同步作战才使得不可一世的法国皇帝遭到了其生命中的“致命打击”。

我们前面说过,线程之间的关系是合作关系。既然是合作,那就得有某种约定的规则,不然合作就会出问题。例如,假如我们有两个线程同时运行,第一个线程在执行了一些操作后想检查当前的错误状态 `errno`,但在其作出检查之前,线程 2 却修改了 `errno`。这样,当第一个线程再次获得控制权后,检查将是线程 2 改写过的 `errno`,这是不正确的,如图 7-2 所示。

之所以出现上述问题,是因为两个原因:

- `errno` 是线程之间共享的全局变量。
- 线程之间的相对执行顺序是不确定的。

因此,要解决上述问题有两个办法,即分别消除上述两个原因。消除第 1 个原因的办法就是限制全局变量,给每个线程一个私有的 `errno`。事实上,我们可以将所有的资源都私有化,让线程之间不共享,那么这种问题就不复存在。

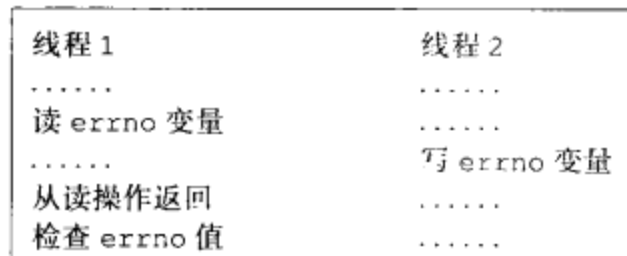


图 7-2 一个进程的两个线程因为操作不同步而造成线程 1 运行错误

但问题是如果所有资源都不共享,那还有必要发明线程吗?甚至也没有必要发明进程了。因为这样就违背了进程和线程设计的初衷:共享资源,提高资源利用率。因此,这种解决办法是不切实际的。

那剩下的办法就是消除第 2 个原因,即让线程之间的相对执行顺序在需要的时候可以确定。在讲述这个办法之前,我们再看一个更加戏剧化的例子。

假定有两个线程 A 和 B,分别执行指令 `x = 1` 和 `x = 2`,即:

线程 A : `x = 1`;

线程 B : `x = 2`;

请问在上述两个线程结束后,`x` 的值是什么?

由于 `x` 是共享变量,且线程之间的相对执行顺序是不确定的,线程 A 既可以在线程 B 之前执行,也可能在线程 B 之后出现。前者将使 `x = 2`,后者则使 `x = 1`。

还有别的结果吗?`x` 有可能等于 3 吗?

一般情况下,在任何计算机体系结构中,`x = 1` 对应的不是一条微指令,即一条高级指令对应的是多条微指令,因此需要多个时钟周期(通常为 6 个时钟周期以上)才能完成。例如,`x = 1` 的赋值语句就是由多个步骤构成。这些步骤可能包括:先把总线清零,然后把 1 加到上面去。这样的话,线程 A 和 B 的执行可能形成如下穿插:A 线程把总线清零,B 线程把总线清零,A 线程将数值 1 加到总线上,B 线程将 2 加到总线上。这样就有可能出现结果 3。当然了,3 的结果的出现依赖于特定的指令集结构。如果指令集结构在执行赋值语句时不是先将总线清 0,然后将要赋值的常数加到总线上,就不会出现结果 3,而只能有结果 1 或者 2。

我们再看一个例子:

线程 A	线程 B
<code>i = 0</code>	<code>i = 0</code>
<code>while (i < 10) {</code>	<code>while (i > -10) {</code>
<code>i++</code>	<code>i--</code>
<code>}</code>	<code>}</code>
<code>printf "A finished"</code>	<code>printf "B finished"</code>

请问，哪个线程会赢呢？即哪个线程会先行运行结束？或者说，有没有哪个线程肯定会赢？答案是不确定的。如果两个线程恰好是你一步、我一步的执行的话，则两个线程都将无法结束。事实上，如果不采取特殊措施，就没有办法确保谁会赢，也没办法确保是否会结束。

由上述例子可见，引入线程后，我们也引入了一个巨大的问题：即多线程程序的执行结果有可能是不可确定的。而不可确定则是我们人类非常反感的東西。那如何在保持线程这个概念的同时，消除其执行结果的不可确定性呢？答案是线程的同步。

7.2 线程同步的目的

线程同步的目的就是不管线程之间的执行如何穿插，其运行结果都是正确的。或者说，我们要保证多线程执行下结果的确定性。而在达到这个目标的同时，要保持对线程执行的限制越少越好。

除此之外，线程同步的另外一个目的涉及到效率。除了我们前面说过的多线程执行的结果是不可确定的外，其执行的效率也是不可确定的。比如说在某段时间内，线程 A 执行了 5 条指令，而 B 只执行了 3 条指令。A 比 B 多执行了 2 条指令。但这并不是问题的关键。问题的关键是到底 A 是否比 B 执行的多，或者是多多少等，皆是不可确定的。如果我们想使其变得确定，就需要进行进程同步。

那么到底什么是“同步”呢？同步就是让所有线程按照一定的规则执行，使得其正确性和效率都有迹可寻。线程同步的手段就是对线程之间的穿插进行控制。下面我们以“金鱼问题”来演示线程同步的各种控制手段。

7.3 锁的进化

读者有没有养过金鱼？养过金鱼的人都知道，金鱼有一个很大的特点，就是没有饱的感觉。因此，金鱼吃东西不会因为吃饱了就停止。它们会不停的吃，一直到鱼缸里面的食物都吃完为止。所以，如果你一直喂，它就一直吃，直到胀死。因此，金鱼吃多少由养金鱼的人来确定，其死活也由人来控制。

现在假定左怡和尤尔两个人合住一套公寓，共同养了一条金鱼。该金鱼每天进食一顿。两个人想把金鱼养活，一天只喂一次，也只能喂一次。如果一天内两人都喂了鱼，鱼就胀死。如果一天内两人都没有喂鱼，鱼就饿死。

他们二人为了把鱼养好，既不让鱼胀死，也不让鱼饿死，做出如下约定：

- 每天喂鱼一次，且仅一次。
- 如果今天左怡喂了鱼，尤尔今天就不能再喂，反之亦然。
- 如果今天左怡没有喂鱼，尤尔今天就必须喂，反之亦然。

显然，要想保证鱼活着，左怡和尤尔得进行某种合作。当然，最简单的情况是不进行任何沟通，每个人觉得需要喂鱼时，查看一下鱼的状态，如果感觉到鱼像是没进过食，则喂鱼，否

则就不喂。图 7-3 给出的是在没有同步情况下左怡和尤尔所执行的程序。

那上述程序里是如何判断 noFeed 呢？程序里没有给出，因此只能依赖于左怡和尤尔的高超养鱼技术：即通过查看鱼的外形看出金鱼当天有没有进食。当然了，这个只有高手才能达到这个水平，一般的人是看不出来的。万一左怡或者尤尔没有看出对方已经喂过鱼了，再喂一次，鱼就胀死了。或者，没有看出对方没有喂过鱼，而没有喂，鱼就饿死了。

左怡：	尤尔：
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}

图 7-3 没有同步情况下左怡和尤尔喂鱼程序

即使假设左怡和尤尔都是养鱼高手，通过查看鱼的外形就能断定鱼是否喂过，上述程序能正确执行吗？答案是否定的。由于线程的执行可以任意穿插，左怡可以先检查鱼，发现没有喂，就准备喂鱼。但就在左怡准备喂但尚未喂的时候，程序切换，轮到尤尔执行。尤尔一看，鱼还没有喂（确实如此），就喂鱼。在喂完鱼后，线程再次切换到左怡。此时左怡从检查完鱼状态后的指令开始执行，就是喂鱼。这样鱼喂了两次，鱼就胀死了，如图 7-4 所示。

事件时序表：	
左怡	尤尔
13:00 观察鱼 (没有喂)	
13:05	观察鱼 (没有喂)
13:10	喂鱼
13:25 喂鱼	
鱼胀死了！	

图 7-4 没有同步情况下左怡和尤尔喂鱼程序的可能结果

为什么这个程序会出现鱼胀死的情况呢？因为左怡和尤尔两个人同时执行了同一段代码。这种两个或多个线程争相执行同一段代码或访问同一资源的现象称为竞争（race）。这个可能造成竞争的共享代码段或资源就称为临界区（critical section）。

当然我们知道两个线程不可能真的在同一时刻执行（单核情况）。但却有可能在同一个时刻两个线程都在同一段代码上。我们这个例子里竞争的是代码，是代码竞争。如果是两个线程同时访问一个数据就叫数据竞争。图 7-3 的程序造成鱼胀死就是因为两个线程同时进入了临界区。

以人类进化来比喻，图 7-3 的程序只相当于氨基酸阶段，胡乱竞争，并不具备任何协调能力。

7.3.1 变形虫阶段

要防止鱼胀死，就需要防止竞争。要想避免竞争，就需要防止两个或多个线程同时进入临界区。要达到这点，就需要某种协调手段。

协调的目的就是在任何时刻只能有一个人在临界区里。这称为互斥（mutual exclusion）。互斥就是说一次只有一个人使用共享资源，其他人皆排除在外，并且互斥不能违反我们前面给出的进程模型。因此，正确互斥需要满足四个条件：

- 不能有两个进程同时在临界区里面。
- 能够在任何数量和速度的 CPU 上正确执行。
- 在互斥区域外不能阻止另一个进程的运行。
- 进程不能无限制地等待进入临界区。

如果任何一个条件不满足，那么你设计的互斥就是不正确的。

那么我们有没有办法确保一次只有一个人在临界区呢？有，让两个线程协调。当然，最简单的协调方法是交谈。问题是左怡和尤尔不见得有时间碰面。那么剩下的办法是留字条。由此，获得第一种同步方式：左怡和尤尔商定，每个人在喂鱼之前先留下字条，告诉对方自己将检查鱼缸并在需要时喂鱼，如图 7-5 所示。

上述机制能否避免鱼胀死呢？不能。如果左怡和尤尔交叉执行上述程序，还是会造成鱼胀死的结局，如图 7-6 所示。

那上面的程序为什么没有效果呢？这是因为我们使用的互斥手段，即留字条，并没有达到互斥的目的。因为字条并没有防止左怡和尤尔两个人同时进入临界区。当然了，与第一个解决方案比起来，本方案还是有所改善，即鱼胀死的概率降低了。只有在左怡和尤尔严格交叉执行的情况下，才可能发生鱼胀死的现象。

此程序虽然加入了一点同步机制，但这个机制太原始，达不到真正的同步目的。以人类进化来比喻，此程序相当于变形虫阶段。

左怡:	尤尔:
if (noNote) {	If (noNote) {
leave note	leave note
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}
remove note	remove note
}	}

图 7-5 第一种同步机制：留字条

事件时序表:	
左怡	尤尔
13:00	检查发现没有字条
13:05	检查发现没有字条
13:10	留字条
13:25	留字条
13:50	观察鱼(没有喂)
14:05	观察鱼(没有喂)
14:10	喂鱼
14:25	喂鱼
鱼胀死!	

图 7-6 第一种同步机制的一种可能结果

7.3.2 鱼阶段

仔细分析发现，上述程序不解决问题的原因是因为我们先检查有没有字条，后留字条。这样造成一个空挡，使得检查字条和留字条之间的空隙。那我们就修改一下顺序，先留字条，再检查有没有对方的字条。如果没有对方的字条，那么就喂鱼，喂完把字条拿掉。不过这种方法需要区分条子是谁的。我们得到如下程序（见图 7-7）。

上述程序怎么样？鱼还会不会胀死呢？答案是不会了。因为无论按照什么顺序穿插，总有一个人的留条子在另一个人的检查字条指令前执行，从而将防止两个人同时进入临界区。因而鱼不会因为两个人都喂而胀死。

那这个程序是成功了。不过，不要过快下结论。我们来看一下，如果两个人穿插执行会出现什么结果。这个时候问题出现了。由于穿插执行，在二者检查字条时，发现对方已经留有字体，从而都不会喂鱼。这个时候鱼饿死，如图 7-8 所示。

虽然胀死很难受，但饿死也好不到哪里去。因此，上述程序也不让我们满意。

本程序虽然防止了胀死，但却允许饿死的情况发生，因此虽然相较于前面两种方式有进步（饿死比胀死好受一点），但还是在很原始的阶段。以人类进化来比，相当于鱼阶段。

左怡:	尤尔:
leave noteZuo	leave noteYou
if (no noteYou) {	if (no noteZuo) {
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}
}	}
remove noteZuo	remove noteYou

图 7-7 第二种同步机制：改进的留字条方法

事件时序表:	
左怡	尤尔
13:10	留字条尤
13:25	留字条左
13:50	检查字条尤(存在)
14:05	检查字条左(存在)
14:10	拿掉字条尤
14:25	拿掉字条左
	没有人喂鱼,鱼饿死!

图 7-8 第二种同步机制的一种可能结果

7.3.3 猴阶段

那么为什么鱼会饿死呢？是因为没有人进入临界区。虽然互斥确保了没有两个人同时进入临界区，但这种没有人进入临界区的情况则有点互斥过了头。要想鱼不饿死，除了互斥外，我们还要保证有一个人进入临界区来喂鱼。那用什么办法来作出这种保证呢？

办法就是让某个人等着，直到确认有人喂了鱼才离去，不要一见到对方的条子就开溜走人。也就是说，在两个人同时留下字条的情况下，必须选择某个人来喂鱼。这样我们就得出第三种同步方式（见图 7-9）。

左怡	尤尔
leave noteZuo	leave noteYou
while (noteYou) {	
do nothing	
}	
	if (no noteZuo) {
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}
	}
remove noteZuo	remove noteYou

图 7-9 第三种同步机制：循环等待

鱼显然不会胀死，因为使用的办法包括了第二种同步方式。那鱼会不会饿死呢？也不会。因为前面说过，鱼饿死的唯一情况是两个人同时留字条，并且又都走人。而上述程序在两个人都留字条的情况下，左怡不会走人，而是一致循环等待直到对方删除字条后，再检查鱼有没有喂，并在没有喂的情况下喂鱼。因此，该同步方式既防止了胀死，又防止了饿死。

我们终于有了个能正确养鱼的程序了，一切似乎大功告成。但真的吗？

我们终于进化到了猴子阶段，能够有一定的组织，不会饿死，也不会胀死了。但这就足够了么？

7.3.4 锁

图 7-9 所描述的第三种同步机制虽然正确，但存在很多问题。

首先是程序不对称。左怡执行的程序和尤尔执行的程序并不一样。那不对称有什么问题吗？当然有。做科学研究的人说完全对称的人最好看。自然，完全对称的程序也最好看。上述程序因为不对称而就不美观。这样就没有达到 Donald Knuth 所谓的“程序就像蓝色的诗歌”的境界。其次，不对称造成程序编写困难。为了追求程序的正确，即使是做同样事情的线程也得编写的不同，这自然就增加编程的难度。最后，不对称还造成程序证明的困难。要想从理论上证明图 7-9 程序的正确性是一件十分复杂的事情。这点研究程序证明的人是很清楚的。

上述程序的另一个大问题是浪费。上述程序中左怡执行的循环等待是一种很大的浪费。但浪费还不是循环等待的唯一问题。它还可能造成 CPU 调度的优先级倒挂。优先级倒挂就是高优先级的线程等待低优先级的线程。例如，如果尤尔先于左怡启动，留下字条后正准备检查是否有左怡的字条时，左怡启动。由于左怡的优先级高于尤尔，左怡获得 CPU，留下字条，进入循环等待。由于左怡的优先级高，尤尔将无法获得 CPU 而完成剩下的工作，进而造成左怡始终处于循环等待阶段无法推进。这样高优先级的左怡就被低优先级的尤尔所阻塞。由于优先级倒挂完全违反了我们设立优先级的初衷，就像总统得听平民指挥似的，因此无法令人容忍（至少无法让总统容忍）。

那我们有没有更好的办法来解决喂养金鱼的问题呢？有，就是继续对我们的同步方案进行改进。那我们在哪一个方案的基础上改呢？自然地我们会想到最后一个方案，因为它已经满足了鱼既不饿死也不胀死条件，无非就是不好看和循环等待。关键是这两点改得了吗？答案是否定的。循环等待不能去掉，一去掉就变成第二个方案；若想使其对称、美观，就需要将尤尔改为和左怡同样，而这样同样会造成鱼饿死的可能。因此对最后一个方案进行修改似乎不是明智之举。看来，我们这种零敲碎打似的推进模式已经走到了尽头。需要新的思路了。

新的思路就是直接对最开始的两个方案进行修改。由于最开始两个方案均达不到既不饿死又不胀死的条件，我们自然选择一个较为美观、简单的方案来修改。两个方案之间，第一个方案完全对称，而第二个方案不是完全对称，因为每个人的条子不同。因此，我们选择第一个方案作为修改的基础。但如何修改呢？

要想知道如何修改，就得知道第一个方案为什么不满足条件。

那第一个方案为什么不满足条件呢？我们说过，是因为检查字条和留字条是两个步骤，中间留有被别的线程穿插的空挡，从而造成字条作用的丧失。我们就想，能否将这两个步骤并为一个步骤，或者变成一个原子操作，使其中间不留空挡，不就解决问题了吗？

换句话说，我们之所以到现在还没把金鱼问题处理掉，是因为我们一直在非常低的层次上打转。因为我们试图工作的层面是鱼和鱼缸这个层面，即留字条是为了防止两个人同时查看鱼和鱼缸。我们仅仅在指令层上进行努力。由于控制的单元是一条条的指令，所以对指令之间的空挡无能为力。而解决这种问题的办法，就是提高抽象的层次，将控制的层面上升到对一组指令的控制。

例如，在金鱼问题里，如果我们将抽象层次从保护鱼和鱼缸的层次提高到保护放置鱼缸的房子的层次，这个问题就可以解决。即设计一种同步措施，使得在任何时候只能有一个

人进入放置鱼缸的房间。这样，检查字条和留字条的两步操作就变成将房间锁上的一个操作。

那么如何保证这个房间一次只进入一个人呢？我们先看看生活当中我们是如何确保一个房间只能进入一个人的。例如，两个老师都想使用同一个教室来为学生补课，怎么协调呢？进到教室后将门锁上，另外一个教师就无法进来使用教室了。即教室是用锁来保证互斥的。那么在操作系统里，这种可以保证互斥的同步机制我们就称为锁。

有了锁，金鱼问题就可以解决了。当一个人进来想喂鱼时，就把放有鱼缸的房间锁住。这样另外一个人进不来，自然无法喂鱼，如图 7-10 所示。

从上面程序我们看到，由于锁的互斥性，左怡和尤尔只能有一个人进入房间来喂鱼，因此鱼不会胀死。并且，如果两人都同时执行上述程序时，由于先拿到锁的人会进入房间来喂鱼，因此鱼也不会饿死。更为重要的是，两个人执行完全同样的代码。既对称，也容易写、证明起来也不困难。这样，金鱼问题得到解决。

以人类进化来比喻，上述程序相当于人的阶段了。

左怡:	尤尔:
lock()	lock()
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}
unlock()	unlock()

图 7-10 锁

锁的基本操作

从图 7-10 可以看出，锁有两个基本操作：闭锁和开锁。闭锁就是将锁锁上，其他人进不来。开锁就是你做的事情做完了，将锁打开，别的人可以进去了。

闭锁操作有两个步骤，分别如下：

- 1) 等待锁达到打开状态。
- 2) 获得锁并锁上。

开锁操作很简单，就是一步：打开锁。

显然，闭锁的两个操作应该是原子操作，即不能分开。不然，就会留下穿插的空挡，从而造成锁的功效的丧失。那么我们是如何让闭锁的两个操作成为一个原语操作的呢？本书将在第 9 章来回答这个问题。这里，我们先来仔细看看锁的机制。

细心的读者可能已经看出，我们给出的第一种解决方案里的字条，从某种意义上来说，就是一把锁，只是这把锁没有将资源锁住。那为什么没有锁住呢？这是因为这把锁不具备一把正常锁所应该具备的特性：

- 锁的初始状态要是打开状态。
- 进临界区前必须获得锁。
- 出临界区时必须打开锁。
- 如果别人持有锁则必须等待。

而我们的字条解决方案违反了第 4 个条件，即在别人持有锁（留下字条）的情况下，也照样进入临界区（因为检查锁是否有别人持有在别人留锁之前进行了）。因此，这个字条无法起到锁的作用，即是一把破锁。

在用了正常的锁之后（见图 7-10），整个程序就可以正确运行了。读者请看，图 7-10 的程序漂亮吧？想形式化证明这个程序也是很容易的事情。

那么这个程序有什么问题没有？如果左怡正在喂鱼的话，尤尔能干什么事情吗？只能等待（等待锁变为打开状态）。如果左怡喂鱼的动作很慢，尤尔等待的时间就会很长。而这种繁忙等待就将造成浪费，也降低了系统效率。那有没有办法消除锁的繁忙等待呢？答案是否定的，因为锁的特性就是在别人持有锁的情况下需要等待。不过我们还是可以减少繁忙等待的时间长度。怎么缩短等待的时间呢？

仔细分析发现，左怡喂鱼并不需要在持有锁的状态下进行。我们就想喂鱼的这段时间不要放在 lock 里面，而是获得锁后留下字条说我喂鱼去了，然后释放锁，再喂鱼。而另一方在拿到锁后先检查有没有字条，有就释放锁，干别的去。没有就留字条，然后释放锁，再喂鱼。这样，由于持锁的时间只有设置字条的时间，因此，对方循环等待的时间会很短，而真正的操作（在这里是喂鱼）则随便多慢也没有问题了。如图 7-11 所示：

左怡：	尤尔：
lock()	lock()
if (noNoteYou)	if (noNoteZuo)
leave noteZuo	leave noteYou
unlock()	unlock()
if (noNoteYou) {	if (noNoteZuo) {
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}
remove note	remove note
}	}

图 7-11 减少锁的繁忙等待时间

图 7-11 比起图 7-10 的程序来说，已经好多了。因为在锁上的繁忙等待时间已经很少了。但不管怎样，终究还是有等待的。那有没有办法不用进行任何繁忙等待呢？有，答案就是睡觉与叫醒，即 sleep&weake up。

7.4 睡觉与叫醒：生产者与消费者问题

什么是睡觉与叫醒呢？就是如果锁被对方持有，你不用等待锁变为打开状态，而是睡觉去，锁打开后再来把你叫醒。我们下面用生产者与消费者的问题来演示这个机制。

有人说这个世界上只存在两种人：生产者和消费者。要么你生产某种东西，要么你消费某种别人生产的东西。当然，你也可能既是生产者，又是消费者。即在一个产品上你是生产者，但在另一产品上你却是消费者。但在某个特定的产品上，一个人只可能是消费者或者生产者，而不能二者同时具有。当然，在这个具体的产品上，你可能既不是消费者，也不是生产者。但对于任何特定的产品，它一定存在一个消费者和一个生产者（我们这里不探讨废品或没有人要的商品）。

那么生产者生产的产品由消费者来消费，但消费者一般不直接从生产者手里获取产品，而是通过一个中间机构，比如商店。生产者把东西往这里放，消费者到这里来拿。为什么需要这个中间机构呢？这是因为商店的存在使得生产者和消费者能够相对独立的运行，而不必亦步亦趋的跟在另一方后面。因为只要商店货架没满，生产者就可以一直生产。他就不用等消费者定一件他才做一件。而如果没有商店，则生产者无法独立操作，必须拿到消费者订单才能生产。而消费者在每次订货后需要等待生产者制造出产品后，才能进行新的订货。这种你走一步我才走一步的生产-消费模式效率十分低下（见图 7-12）。



图 7-12 在现实生活中，超市相当于消费者和生产者之间的缓冲区

用计算机来模拟生产者和消费者是很简单的事：一个进程代表生产者，一个进程代表消费者，一片内存缓冲区代表我们的商店。生产者生产的物品从一端放入缓冲区，消费者从另外一端获取物品，如图 7-13 所示。



图 7-13 生产者-消费者问题

一个非常好的例子是校园的售货机。售货机是缓冲区，负责装载售货机的送货员是生产者，而购买可乐、糖果的学生自然就是消费者。只要售货机不满也不空，则送货员和学生可以继续他们的送货和消费。问题是，如果学生来买可乐，却发现售货机空了，怎么办？学生当然有两个选择：一是坐在售货机前面等着，直到送货员来装货为止；二是回宿舍睡觉，等售货员来了后再来买。第一种方式显然效率很低，估计没有什么人愿意这么做。相比较起来，第二种方式要好些。只不过睡觉中的学生不可能知道售货员来了，因此我们需要送货员来了后将学生叫醒。

同样，如果送货员来送货发现售货机满时也有两种应对办法：一是等有人来买掉一些东西，然后将售货机填满；二是回家睡觉，等有人买了后再来送货。当然，这个时候买者需要将

送货员叫醒。以程序来表示生产者和消费者问题的解决方案，如图 7-14 所示。

图 7-14 中的 sleep 和 wakeup 就是操作系统里的睡觉和叫醒操作原语。一个程序调用 sleep 后将进入休眠状态，其所占用的 CPU 将被释放。一个执行 wakeup 的程序将发送一个信号给指定的接收进程，如 wakeup(producer) 就是发送一个信号给生产者。

```

#define N 100      缓冲区的大小
Int count = 0;    缓冲区项目数

void producer(void)
{
    Int item;
    While(TRUE) {
        Item = produce_item();
        If(count == N) sleep();
        Insert_item(item);
        count = count + 1;
        If(count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    Int item;
    While(TRUE) {
        If(count == 0) sleep();
        Item = remove_item();
        count = count - 1;
        If(count == N - 1) wakeup(producer);
        Consume_item(item);
    }
}

```

图 7-14 消费者 - 生产者的同步程序

我们仔细来看上面的程序。最上面两行定义了缓冲区的大小（可容纳 100 件商品）和当前缓冲区里商品个数，初始化为 0。生产者程序的运行如下：生产一件商品，检查当前缓冲区的商品数，如果已经满，则进入睡眠状态；否则将商品放入缓冲区，将计数加 1。然后判断计数是否等于 1，如果是，说明在放这件商品前缓冲区为 0，有可能存在消费者来见到空缓冲区而去睡觉，因此需要发送叫醒信号给消费者。

消费者程序运行如下：先检查当前商品计数，如果是 0，没有商品，当然睡觉去。否则，从缓冲区拿走一件商品，将计数减 1。然后判断计数是否等于 N - 1，如果是，说明在拿这件商品前缓冲区为 N，有可能存在生产者来见到满缓冲区而去睡觉，因此需要发送叫醒信号给生产者。然后尽情地享用商品。

这个程序看上去似乎正确无误。但真是这样吗？

可能有的读者会注意到一个问题：如果生产者在判断 count == 1 时发送叫醒信号给消费者，但也有可能没有消费者在睡觉（售货机空的时候，恰恰没有人来买东西，自然就不会有学生睡觉了），这个信号不是无的放矢吗？同理，消费者也存在无的放矢发送叫醒生产者信号的问题。那这是不是问题呢？

打过电报的人都知道，如果对方因故（如查无此人、此人暂时不在）无法接收电报，并不会造

成什么危害,了不起就是大家浪费一点时间来收发这封电报而已。这里也是一样,如果发送的信号没有进程接收,则这个信号权当浪费了,并不会造成任何损害。因此,这个可能无的放矢的信号发送不是什么问题。

那还有没有其他问题呢?

我们看出来了,这个 `count` 有问题。因为变量 `count` 没有被保护,可能发生数据的竞争。即生产者和消费者可能同时对该数据进行修改。例如,假定 `count` 现在等于 1,生产者先运行,对 `count` 加 1 操作后 `count` 变为 2,但在判断 `count` 是否等于 1 之前,CPU 被消费者获得,随后对 `count` 进行了减 1 的操作后切换回生产者,这个时候 `count` 等于 1,因此生产者将发出叫醒消费者的信号。显然,这个信号是不应该发出的。

对 `count` 没有保护并不是上述程序的唯一问题。关键问题是上述程序可能造成生产者和消费者均无法往前推进的景况,即所谓的死锁。例如,我们假定 `consumer` 先来,这个时候 `count = 0`,于是睡觉去,但是在判断 `count` 等于 0 后却在执行 `sleep` 语句前 CPU 发生切换,生产者开始运行,它生产一件商品后,给 `count` 加 1,发现 `count` 结果为 1,因此发出叫醒消费者信号。但这个时候 `consumer` 还没有睡觉(正准备要睡),所以该信号没有任何效果,浪费了。而生产者一直运行直到缓冲区满了后也去睡觉。这个时候 CPU 切换到消费者,而消费者执行的第 1 个操作就是 `sleep`,即睡觉。至此,生产者和消费者都进入睡觉状态,从而无法相互叫醒而继续往前推进。系统死锁发生。

那我们如何解决上述的两个问题呢?对第 1 个问题,解决方案很简单:用我们刚刚讲完的锁!在进行对 `count` 的操作前后分别加上 `lock` 和 `unlock` 即可防止生产者和消费者同时访问 `count` 情况的出现。不过有细心的读者会说,我们不就是因为锁存在繁忙等待才发明 `sleep&wakeup` 原语吗?怎么又把锁给请回来了呢?

确实,我们不喜欢锁所用的繁忙等待,因而发明了 `sleep&wakeup` 原语。这样在需要等待时,我们去睡觉。但是,我们不喜欢等待,并不是一点都不能等待。只要等待的时间很短,我们是可以接受的。就是让一个人等 1 分钟,通常都不会觉得漫长。而在 `count` 的访问前后加上锁所造成的繁忙等待是很短的。不就是将商品放入或拿出缓冲区吗?这要多长时间呢。

好,上述解释勉强算把第 1 个问题解决了。但第二个问题怎么解决呢?

显然,生产者和消费者不会自己从睡觉中醒过来。所以如果二者同时睡觉去了,自然也无法叫醒对方。那解决的方案就是不让二者同时睡觉。而造成二者同时睡觉的原因是因为生产者发出的叫醒信号丢失(因为消费者此时还没睡觉)。那我们就想,如果用某种方法将发出的信号累积起来,而不是丢掉,问题不就解决了吗?在消费者获得 CPU 执行 `sleep` 语句后,生产者在这之前发送的叫醒信号还保留,因此消费者将马上获得这个信号而醒过来。而能够将信号累积起来的操作系统原语就是信号量。

7.5 信号量

信号量可以说是所有原语里面功能最强大的。它不光是一个同步原语,还是一个通信原语。而且,它还能作为锁来使用!本章前面已经讨论过作为通信原语的信号量,现在我们来看其作为

同步原语和锁的能力。

semaphore 说白了就是一个计数器。其取值为当前累积的信号数量。它支持两个操作：加法操作 Up 和减法操作 Down，分别描述如下：

Down 减法操作：

- 1) 判断信号量的取值是否大于等于 1。
- 2) 如果是，将信号量的值减去 1，继续往下执行。
- 3) 否则，在该信号量上等待（线程被挂起）。

Up 加法操作：

- 1) 将信号量的值增加 1（此操作将叫醒一个在该信号量上面等待的线程）。
- 2) 线程继续往下执行。

这里提请读者注意的是，Down 和 Up 两个操作虽然包含多个步骤，但这些步骤是一组原子操作，它们之间是不能分开的。如何实现原子操作我们将在第 9 章里论述。

Up 和 Down 操作在历史上称为 P 和 V 操作。P 和 V 指的是荷兰语里 *proberen* 和 *verhogen* 两个单词，分别表示增加和减少的意思。由于美国人在操作系统领域取得了绝对控制权，自然不能用什么荷兰语来表示操作系统里面最重要同步原语的两个基本操作，因此，将名字改为 Up 和 Down。

如果我们将信号量的取值限制为 0 和 1 两种情况，则我们获得的就是一把锁，也称为二元信号量（Binary Semaphore），其操作如下：

二元信号量 Down 减法操作：

- 1) 等待信号量取值变为 1。
- 2) 将信号量的值设置为 0。
- 3) 继续往下执行。

二元信号量 Up 加法操作：

- 1) 将信号量的值设置为 1。
- 2) 叫醒在该信号量上面等待的第 1 个线程。
- 3) 线程继续往下执行。

使用二元信号量进行互斥的形式如下：

```
down ()
<临界区>
up ()
```

由于二元信号量的取值只有 0 和 1，因此上述程序防止任何两个程序同时进入临界区。二元信号量还可以用来管理线程的执行顺序，例如，下述程序保证 B 将在 A 之前执行完毕：

- 初始化信号量为 0
- A 进程 B 进程
- down () 做工作...
- 继续执行... up ()

二元信号量具备锁的功能，实际上它与锁很相似：Down 就是获得锁，Up 就是释放锁。但

它又比锁更为灵活：因为等在信号量上的线程不是繁忙等待，而是去睡觉，等另外一个线程执行 Up 操作来叫醒。因此，二元信号量从某种意义上说就是锁和睡觉与叫醒两种原语操作的合成。

有了信号量，我们就可以轻而易举地解决生产者和消费者的同步问题。具体说来如下：我们先设置三个信号量，分别如下：

- mutex:
 - 一个二元信号量，用来防止两个线程同时对缓冲区进行操作
 - 初值为 1
- full:
 - 记录缓冲区里商品的件数
 - 初值为 0
- empty:
 - 记录缓冲区里空置空间的数量
 - 初值为 N（缓冲区大小）

我们的生产者和消费者程序，如图 7-15 所示。

```

#define N 100          /* 定义缓冲区大小 */
typedef int semaphore; /* 定义信号量类型 */
semaphore mutex = 1;   /* 互斥信号量 */
semaphore empty = N;   /* 缓冲区计数信号量,用来计数缓冲区里的空位数量 */
semaphore full = 0;    /* 缓冲区计数信号量,用来计数缓冲区里的商品数量 */

void producer(void)
{
    int item;
    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;
    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

```

图 7-15 生产者和消费者程序

该程序解决了我们前一个版本的问题吗？很显然，上述程序中生产者和消费者不可能同时睡觉而造成死锁。因为两个人同时睡觉就意味着：full = 0 (producer 才睡觉)，并且 empty = 0 (consumer 睡觉的条件)。那么 empty 和 full 能够同时为 0 吗？当然不会，因为初值是 empty = N 而 full = 0。要使 empty 等于 0，生产者就必须生产，而一旦生产者开始生产，full 就不能为 0 了。所以两个不会同时睡觉。

这样上述程序既保护了缓冲区不会被生产者和消费者同时访问，又防止了生产者或消费者发送的信号丢失。生产者生产了多少商品，信号量 full 就取多大的值，这就相当于我们前一个版本里面的发送的信号个数。而因为消费者等待的地方就是 full 这个信号量，因此，生产者

生产了多少商品,就可以最多这么多次叫醒消费者。反之亦然,消费者消费了多少商品,信号量 `empty` 里就记录了多少数量,也就可以多少次叫醒生产者。这样解决了信号丢失问题。

到现在我们可以问一个问题了,就是为什么我们需要三个信号量呢? 一个二元信号量用来互斥,一个信号量用来记录缓冲区里商品的数量不就可以了么? 缓冲区里空格的数量不是可以由缓冲区大小和缓冲区里商品的数量计算得出吗? 干吗需要一个 `full` 和一个 `empty` 来记录满的和空的呢? 这是因为生产者和消费者等待的信号不同,它们需要睡在不同的信号上。

7.6 锁、睡觉与叫醒、信号量

到现在我们看到,锁解决了同步问题,但带来的是循环等待,我们不满意。为了消除循环等待,我们发明了睡觉与叫醒。但睡觉与叫醒又带来了死锁,因此,我们发明了信号量。而锁的出现本章前面已经花了大量篇幅进行了介绍。由此可以看出,操作系统的各种原语操作并不是毫无联系的,而是一环扣一环,具有严密的逻辑连贯性。

那么信号量是否就是我们的终极原语了呢? 它是否就没有任何问题了呢?

我们先来看一下,在图 7-12 的程序中,两个 `down` 的操作如果顺序颠倒的话,有什么问题? 例如,我们将消费者的两个 `down` 操作颠倒一下,即先 `down mutex`,然后 `down full`。假定 `consumer` 来了,先 `down mutex`,因为 `mutex` 此时取值为 1,这个 `down` 操作将通过,并且 `mutex` 的取值将变为 0。消费者然后 `down full`,由于 `full` 目前的值为 0,于是消费者等在 `full` 这个信号量上,睡觉了。这个时候生产者来了,它先 `down empty`,发现 `empty` 是 `N`,于是顺利通过,接着 `down mutex`,但是 `mutex` 的取值此时是 0,因此生产者将等在 `mutex` 上。这样生产者等待消费者释放 `mutex`,而消费者则等待生产者 `up full`。两个人均无法继续推进,死锁产生。

同理,如果将生产者的两个 `down` 操作颠倒一下,也同样会产生死锁。

但如果我们将两个 `up` 操作颠倒一下,有什么问题吗? 例如,将生产者的两个 `up` 操作颠倒一下,先 `up full` 再 `up mutex`。由于信号量能够记住所有的信号,颠倒 `up` 操作将不会改变程序的正确性。但却可能使得程序的效率下降。例如,在做了上述修改后,生产者在 `mutex` 之间的临界区里面的操作增加了,这样,所有等待 `mutex` 的进程的等待时间就延长了。另外,如果系统在生产者 `up full` 之后,切换到叫醒的消费者,那么消费者叫醒后又将重新在 `mutex` 上等待。这样也就多了一层开销。这些延时和开销在只有两个信号量时还不太会觉察到,但信号量多了,这种效率的下降就会明显。也许有的读者认为这点效率的不同无关紧要,那下面的这个真实故事将改变读者的看法。

在 90 年代前期,美国电话电报公司 AT&T 一直独霸美国的通信行业。绝大部分美国人均使用 AT&T 作为自己的电话服务公司。而后来者 MCI(后来变为环球电信 WorldCom)公司成立后,就面临着如何从 AT&T 公司抢夺客源的问题。自然,价格战是免不了的。但光靠价格战是不够的。MCI 于是做起了研究。他们发现一个顾客拿起电话,拨完号后,等待对方应答的耐心是 4 秒钟。如果 4 秒钟内没有人应答,则 75% 的人会将电话挂断,不打了。MCI 的进一步研究发现,AT&T 的长途电话接通时间几乎总是超过 5 秒钟,有时候甚至要等十几秒钟才接通。这样 MCI 的机会就来了。那就是在 4 秒钟时间内接通顾客的电话。

但问题是,4秒钟接通顾客的电话做得到吗?从顾客拨号开始,就需要在数据库内进行查找,将被叫方的地址和所在服务局的信息调出来,并与主叫方的地址和服务局信息进行关联,选出最佳的路由,然后接通电话,这需要速度非常高的存储系统的支持。而市面上的存储系统都无法满足在小于4秒钟内读取大量数据的苛刻要求。MCI 找来找去,终于找到了一家公司 EMC,其麾下的 Symmetrix 高速智能存储系统满足其要求。就这样,MCI 因为接通时间短于用户的心理等待极限而获得了大块的市场,EMC 公司也因为能够生产高速存储器而称为 MCI 的最大存储设备供应商。由此可见,效率的高低有时候是决定性的因素。

由此我们也可以看出,使用信号量原语时,信号量操作的顺序至关重要。稍有不慎,就可能发生死锁。而这还是在我们只有三个信号量的情况。如果一个程序使用十个、几十个信号量,程序员将很难搞清楚正确的顺序到底是什么,而写程序将变成一个巨大的挑战。事实上,如果一个程序的信号量繁多,死锁或者效率低下几乎是可以肯定的。

那有没有办法改变这种状况,使得编程序不是那么大的一个挑战呢?有办法。办法就是管程。

7.7 管程

前面说过,信号量存在程序编写困难或程序效率低下的问题。那我们就想,如果能够将信号量的这些组织工作交给一个专门的构造来管,程序员不就解脱了吗?于是我们发明了管程。管程的英文单词是 Monitor,即监视器的意思。它监视的就是进程或线程的同步操作。

管程是一个程序语言级别的构造,即它的正确运行由编译器负责保证。这就是计算机里面的一条哲学原理:你不行的时候,把困难交给别人。

具体来说,管程就是一组子程序、变量和数据结构的组合。言下之意,你把需要同步的代码用一个管程的构造框起来,将需要保护的代码置于 begin monitor 和 end monitor 之间,即可获得同步保护。在任何时候只能有一个线程活跃在管程里面。那谁来保证这一点呢?编译器。编译器在看到 begin monitor 和 end monitor 时知道其间的代码需要同步保护,在翻译成低级代码时就会将需要的操作系统原语添上,使得两个线程不能同时活跃于同一个管程内。图 7-16 描述的就是一个管程的例子。

```
begin monitor
  integer i;
  condition c;
  procedure producer();
  ...
end;

  procedure consumer();
  ...
end
end monitor
```

图 7-16 管理

在管程里面,使用两种同步机制:锁用来互斥,条件变量用来控制执行的顺序。从某种意义上说,管程就是锁加上条件变量。那么什么叫条件变量?条件变量就是线程可以在上面等待的东西,而另外一个线程则可以通过发送信号将在条件变量上等待的线程叫醒。因此,条件变量有点像信号量,但又不是信号量,因为不能对其进行 Up 和 Down 操作。

管程的中心思想是运行一个在管程里面睡觉的线程。但是在睡觉前需要把进入管程的锁或信号量释放,否则在其睡觉后别的线程将无法进入管程,就会造成死锁。本书前面说过,在临

界区里面做的事情要越少越好，那自然不能在里面睡觉了。但这里恰好相反。这是因为，在正常情况下，只有一个线程在临界区里，因此，在临界区待的时间越长，别的线程的等待时间就越长。但在这里情况就不一样了。因为允许别的线程进入管程，因此我们可以睡觉。

而实现锁的释放和睡觉这两件事情必须是原子操作，即中间不能有空挡，否则将造成两个线程同时活跃在管程里，这样就违反了我们关于管程的约定。当然了，这种违反并不会造成程序错误，因为其中的一个线程的下一步操作是睡觉，不会与另外的线程争夺共享资源。

下面我们来看如何使用管程来实现消费者生产者的同步（见图 7-17）。我们在管程内部定义了两个操作：生产者往缓冲区插入商品的操作，和消费者从缓冲区取出商品的操作。由于这两个操作使用共享资源：缓冲区，因而需要同步。故此我们将其放在管程里面。

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert (item: integer);
  begin
    if count == N then wait(full);
    insert_item(item);
    count ++;
    if count == 1 then signal(empty);
  end;
  function remove: integer;
  begin
    if count == 0 then wait(empty);
    remove = remove_item;
    count --;
    if count == N-1 then signal(full);
  end;
  count := 0;
end monitor;
```

图 7-17 生产者消费者的管程内部部分

上图中我们看到两个条件变量 full 和 empty;但没有看到锁。那锁在哪里呢？我们前面说过，管程的互斥保证由编译器负责，即编译器在编译上述代码段时将把与锁有关的部分加上。因此，应用程序员不用担心这个问题。

管程里面的两个操作 wait 和 signal 的语义则分别如下：

wait(x)以原子操作完成下述三个步骤：

- 1) 释放锁。
- 2) 将本线程挂在条件变量 x 的等待队列上。
- 3) 睡觉，等待被叫醒。

signal 则与我们前面讲过的一样，将等在指定条件变量上面的第 1 个线程叫醒。在叫醒方面，管程还提供另外一个所谓的广播原语(broadcast)，其语义是将指定条件变量上面的所有等待线程全部叫醒。

这里需要读者注意的是，在一个线程调用 wait、signal 或者 broadcast 之时，该线程必

须持有与管程相连的锁。

由于生产者对缓冲区进行操作的函数 `insert` 和消费者对缓冲区进行操作的函数 `remove` 均处于管程 `ProducerConsumer` 里面,而管程又具有自动防止两个或多个线程同时活跃于其内的功能,因而,缓冲区的访问,即缓冲区计数器 `count` 的修改都将是互斥的。

生产者生产商品的部分和消费者消费商品的部分处于缓冲区活动之外。生产者在生产出一件商品后,调用处于管程里面的 `insert` 函数来将商品放入缓冲区。而消费者则先调用管程里面的 `remove` 函数从缓冲区里面获取一件商品,然后在管程外面慢慢消费商品。图 7-18 则显示的是生产者和消费者在缓冲区外活动的部分。

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    end
end;
```

图 7-18 生产者消费者的管程外部部分

下面我们仔细来分析一下在生产者和消费者在缓冲区的工作流程。假如生产者先开始运作,生产出一件商品,然后调用 `insert` 函数。`insert` 函数首先判断 `count` 是否等于 `N`,如果等于 `N`,则将自己挂在条件变量 `full` 的等待队列上,线程将切换到消费者。否则执行下一步操作将商品放入缓冲区,缓冲区计数 `count` 加 1。然后判断 `count` 是否大于等于 1,如果是,发出信号到条件变量 `empty` 上。此时,如果有消费者在该变量上等待,则将被叫醒。

如果消费者来到,它首先调用 `remove` 函数。`remove` 函数先判断 `count` 是否等于 0。如果是就将自己挂在 `empty` 条件变量的等待队列上,线程将切换到生产者。如果不是则执行下一步,将商品从缓冲区取出,计数器减 1。然后判断计数器是否小于等于 `N - 1`。如果是,发送信号将可能在 `full` 上面等待的生产者叫醒。

乍一看,这个解决方案与本章前面介绍过的 `sleep` 和 `wakeup` 非常相似:只不过在那里,我们用的是 `sleep` 而不是 `wait`,是 `wakeup` 而不是 `signal`。那这个方案是否也存在 `sleep` 和 `wakeup` 方案的缺点,也就是死锁呢?

答案是否定的。这是因为,在使用睡觉和叫醒原语实现生产者和消费者同步的方案中,没有锁的使用。这样,在一个线程判断是否需要睡觉和真的去睡觉这两个操作中间存在空挡,从而对方可以在此时切入,造成信号的丢失,导致死锁的可能。但在管程机制下,这种空挡没有了。虽

然判断是否需要等待和等待是两个语句,但请记住:进入管程需要获得该管程的锁(由编译器提供),因此,在一个准备等待的线程真正进入等待之前,由于其持有的管程锁尚未释放,另外的线程是无法执行管程里面的任何代码的。而在锁释放之时,相关线程已经进入等待状态。此时如果一个线程发送信号,自然将被收到,而不会丢失。

这里提醒读者注意:一旦一个线程发出释放等待线程的 signal,则此时将有两个线程同时活跃于管程内。而这,违反了我们对于管程的约定。为了防止这个问题,管程机制特别约定:signal 语句是一个线程在管程里面执行的最后一个操作。这样,即使理论上有两个线程同时活跃于管程内,但实际上只有一个线程活跃。因为另一个线程的下一步操作已经在管程之外。我们关于管程的约定从而得到维持。

MESA 和 HOARE 管程

前面说过,当一个线程发出 signal 信号后,在理论上将有两个线程同时活跃在管程内。但我们知道,管程只有一把锁。如果两个线程同时活跃,那谁将持有管程的锁呢?

自然的想法是,由于发送 signal 的线程执行的是管程里面的最后一个语句,不如让其继续执行,从而转到管程外面(因为其下一步操作就是在管程外),这样管程里面不是还只有一个线程吗?即叫醒者继续持有锁,并在离开管程时将锁释放,此时被叫醒者将获得管程的锁,从而可以继续执行。这种处理方法自然,但是保守。因为如果这样的话,线程就没有必要提前叫醒等待的线程。为什么不等到执行到管程外面再叫醒呢?

第二种办法是给被叫醒者优先级,即在发送 signal 时同时释放锁,让被叫醒者获得锁。从而在 signal 后,第一个运行的线程将是被叫醒的线程。叫醒者只能在被叫醒者运行完毕或因其他原因释放锁之后才能继续运行。这种方式因为是 HOARE 提出,因此这种给予被叫醒者优先的管程称为 HOARE 管程。

前面两种方法都是在管程设计时就确定了谁将有优先权:前者将优先权给了叫醒者,后者则将优先权给予被叫醒者。这两种方法因为都是在管程设计时就确定了,十分的不灵活,它限制了操作系统的作用。对于操作系统设计人员来说,我们希望下一步谁执行由操作系统说了算。即让两个线程竞争这把锁。这样,操作系统就可以在竞争中发挥作用,使用各种机制动态地调整每个线程获取锁的优先级。这种管程就称为 MESA 管程。

MESA 管程的 signal 处理方式如下:叫醒者在发出 signal 后释放锁;被叫醒者与叫醒者同时竞争这把锁。谁先获得锁,谁先执行。

由于 MESA 管程给予了操作系统以重要角色,它获得大多数操作系统的认可。

7.8 消息传递

那么管程有什么问题没有?

有。管程最大的问题是对编译器的依赖。因为我们需要编译器将需要的同步原语加在管程的开始和结尾。而这是一个令操作系统人员不放心的选择。俗话说,“相信别人,就等着灾难的

发生吧。”另外,搞编译的人也不想在这上面花费心血,他们有编译方面的许许多多的问题还没有解决。而且在实际上,多数的程序设计语言也并没有实现管程机制。

另外,管程只能在单台计算机上发挥效果。如果想在多计算机环境下(或者网络环境下)进行同步,那就需要别的一种机制了。这种别的机制就是消息传递。

消息传递是通过同步双方经过互相收发消息来实现。它有两个基本操作,发送 send 和接收 receive。它们均是操作系统的系统调用,而且既可以是阻塞调用,也可以是非阻塞调用。

```
- Send(destination, &message)
- Receive(source, &message)
```

而同步需要的是阻塞调用。即如果一个线程执行 receive 操作,就必须等待收到消息后才能返回。也就是说,如果调用 receive,则该线程将被挂起,在收到消息后,才能转入就绪。

图 7-19 描述的是使用消息传递实现的生产者和消费者同步问题:

```
#define N 100                                /* 缓冲区的大小为 100 */
void producer(void)
{
    Int item;
    message m;                                /* 消息框 */
    While(TRUE) {
        Item = produce_item();
        receive(consumer, &m);                /* 等待空消息框 */
        Build_message(&m, item);              /* 构造一个消息 */
        Send(consumer, &m);
    }
}

Void consumer(void)
{
    Int item;
    message m;
    for(i=0; i<N; i++) send(producer, &m);    /* 发送 N 个空消息框 */
    While(TRUE) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        consume_item(item);
    }
}
```

图 7-19 使用消息传递实现的生产者和消费者同步

在图 7-19 中,生产者每生产出一件商品,就需要从消费者那里获取一个空的盒子(m),然后将产品装进盒子里,再把装了产品的盒子发送给消费者。消费者的工作过程刚好反过来,先发送 N 个空盒子给生产者;然后等待生产者将生成的商品发送过来,消费了之后,将空盒子发送过去。而只要当前既有空盒子,又有满盒子,则生产者和消费者都可以独自运行。如果所有盒子都满了,生产者将在执行下一个 receive 操作时阻塞,直到消费者消耗掉一件商品并发回空盒子为止。如果在某个时候盒子全空了,则消费者在执行下一个 receive 时将阻塞,直到生产者生成出至少一件商品并发过来一个满盒子为止。

生产者和消费者就这样通过消息的传送进行同步,既不会死锁,也不会繁忙等待。而且,无

需使用临界区等机制。更为重要的是,它可以跨计算机进行同步,即可以对处于不同计算机上的线程实现同步。由于这些优点,消息传递是当前使用非常普遍的线程同步机制(当然了,它更是一种通信机制,记得前面讲过的消息队列吗?)。

那么消息传递有什么问题没有?有。最大的问题就是消息丢失和身份识别。消息在一台计算机内部传递时丢失的可能很低,但在网络间传输时丢失的可能性就很大了,这是因为网络的不可靠性所致。而身份识别指的是你怎么确定收到的信息就是从你想要的对象那里发出的呢?

当然,通过设计各种网络协议,如 TCP 协议,我们可以将网络数据传输的可靠性提高。但即使是 TCP,也不是 100% 可靠。身份识别则可以使用诸如数字签名和加密等技术来弥补。

使用消息传递的另外一个缺点就是效率。往返发送消息存在系统消耗。另外,数据传输也存在延迟。如果网络速度很慢怎么办呢?

7.9 栅栏

栅栏 barrier,是本章最后要讲的一个通信原语。顾名思义,栅栏就是一个障碍。到达栅栏的线程必须停止下来,直到栅栏被除去才能往前推进。该原语主要用来对一组线程进行协调。因为有时候一组进程协同完成一个问题,我们需要所有进程都到同一个地方汇合之后一起再向前推进。例如,在进行并行计算时就会遇到此种需要,如图 7-20 所示:

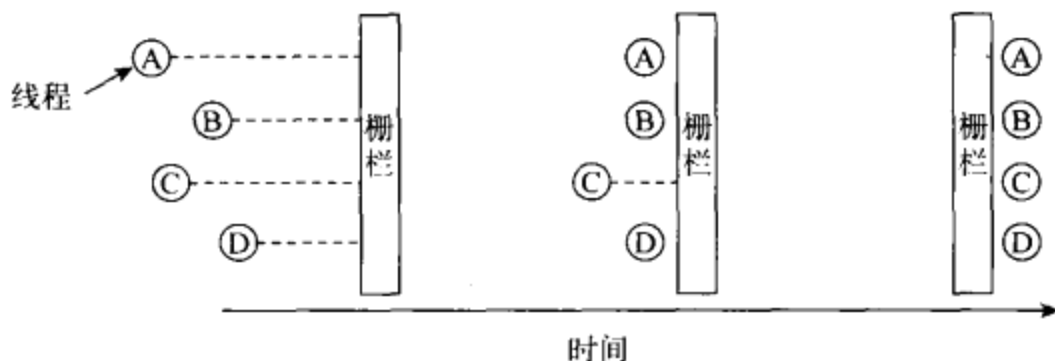


图 7-20 栅栏(来源:参考文献[3])

如矩阵乘法。在不优化的情况下,矩阵乘法的时间复杂性 N^3 (N 为矩阵维数),而 Strassen 的优化算法则是将矩阵乘法分解为 4 个小矩阵的乘法,从而达到 $n^{2.71}$ 效率。而这个算法的思想是先计算出 4 个小矩阵乘法的结果,然后通过加减运算获得最后的结果。如果将 4 个小矩阵乘法用 4 个线程来执行,则最后的加减运算必须等待 4 个线程的乘法运算都结束后才能往前推进。而等待 4 个线程都到达同一个状态的机制就可用栅栏。

思考题

1. 互斥应该满足什么条件?
2. 线程可以在管程里面睡觉吗?为什么?
3. 请列出各种同步原语出现的时间逻辑顺序。
4. 有一个同学提出了一种不使用信号量而又可以解决生产者 - 消费者问题中

sleep&wakeup 原语中发生的死锁。该方法就是再加一把锁将“检查是否睡觉”和“睡觉”两个语句变为一组原子操作,从而防止信号丢失而造成死锁。即

```

void producer(void)          void consumer(void)
...
lock()                        lock()
    If(count == N) sleep();    If(count == 0) sleep();
unlock                        unlock()
...                           ...

```

请问该方法可行吗?

5. 简要论述锁、睡觉与叫醒和信号量之间的逻辑关系。
6. 假定在进程通信中使用一个类似邮箱的机制。当一个进程试图往一个满箱放东西的时候,或者从一个空箱拿东西的时候,该进程并不阻塞。而是得到一个错误返回码。该进程立即重复上述过程(往邮箱里面放东西或从邮箱拿东西),直到成功为止。请问这种机制会造成竞争吗?为什么?
7. 有一个同学认为本书给出的管程模式存在很大缺陷。于是他提出了改进:将 wait、signal 和 broadcast 三种原语去掉,代之以一种新的原语:waituntil。该新原语有一个布尔表达式作为参数,例如,waituntil $x + y < z$ 。而锁的功能不变。请问你对这种改进的观点如何?
8. 下面的程序使用原子加载和存入操作来实现互斥:

```

/* 全局变量 */
blocked[0] = false; blocked[1] = false; turn = 0;

/* 线程 0 */
while(1){
    blocked[0] = true;
    while(turn != 0){ while(blocked[1]) do; turn = 0; }
    // critical section
    blocked[0] = false;
}

/* 线程 1 */
while(1){
    blocked[1] = true;
    while(turn != 1){ while(blocked[0]) do; turn = 1; }
    // critical section
    blocked[1] = false;
}

```

请问该实现正确吗?如果正确,请证明。否则,请给出反例。

9. 假定 n 个线程要访问 m 个独立的共享对象。而你必须保证没有数据腐败发生。你保证的方法就是对共享数据的所有访问上锁。
 - a) 请问防止数据腐败所需要的最少数量的锁是多少?
 - b) 如果要获得最大的并发性,需要多少锁?
 - c) 最大化并发性能否降低系统性能?

10. 共享卫生间

某单位为节省经费,决定建造男女共用的单性卫生间。为满足社会风化要求,卫生间的使用需要满足如下条件:在任何时候不同性别的人不能同时在卫生间里。

你的任务就是写一个程序来模拟卫生间的使用。你可以使用的工具是 Mesa 管程。你需要编写下述 4 个函数:

```
woman_wants_to_enter(),
man_wants_to_enter(),
woman_leaves(),
man_leaves().
```

你在这些函数里面可以使用原语 lock(), unlock(), signal(), wait() 和 broadcast() 来控制对卫生间的使用。我们假定同一性别的人进入卫生间的数量不受限制。

11. 另外版本的卫生间共享

重做第 10 题的卫生间共享问题。这次把优先级赋予当前使用卫生间的性别。例如,如果卫生间里面已经有女人再使用,新来的女人即可以直接进去,即使有男人等在卫生间外面。

12. 第 3 个版本的卫生间共享

重做上述问题,这次需要保证公平,并防止饥饿:如果女人在卫生间,则只要没有男人等在外面,新来的女人皆可以进到卫生间。如果有男人等待,则新来的女人就不能进去,而必须等在男人后面。当卫生间里的最后一位女人离开时,等待的男人可全部进入。

13. 读者写者问题

中国航信的航空订票数据库系统是中国国内所有航空公司的共享数据库系统。所有的旅行社订票均使用该系统进行查询、预定和出票。由于旅行社众多,在任意一个时间都可能多个进程对该数据库进行操作。所有读写数据库的进程分为读者和写者:读者读数据库而已,写者则对数据库进行更新。为保持数据一致性,该数据库的访问需满足如下限制:

- 多个读者可以同时数据库进行(读)操作。
- 如果有一个写者进程在对数据库进行(写)操作,则其他进程都不能对数据库进行操作。

请写出读者和写者的伪代码程序。

第8章 进程调度

引子

- 11:4 他们说：“来吧，我们要建造一座城和一座塔，塔顶通天，为的是要传扬我们的名，免得我们分散在地上。”
- 11:5 耶和华降临，要看看世人所建造的城和塔。
- 11:6 耶和华说：“看哪！他们成为一样的人，使用一样的言语，如今既做起这事来，以后他们所要做的事就没有不成的了。”
- 11:7 “我们下去，在那里变乱他们的口音，使他们的言语彼此不通。”
- 11:8 于是，耶和华使他们从那里分散在地上，他们就停工不造城了。
- 11:9 因为耶和华在那里变乱天下人的言语，使众人分散在地上，所以那城名叫巴别（注：就是“变乱”的意思，如图8-1）。

——摘自《创世纪》11章4~9节

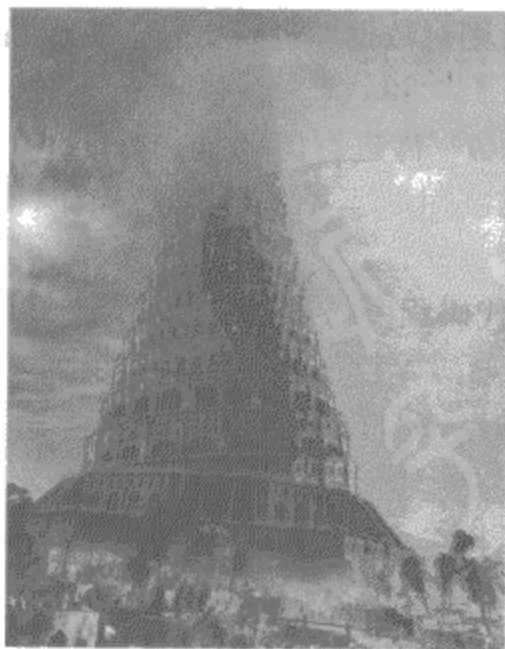


图8-1 传说中人类建造的巴别塔

神通过调度驱散了聚集在一起的人类,而达到自己的目的。对于掌管着众多进程线程的操作系统来说,也需要通过调度、驱散或聚集进程,来达到维持计算机功能的目的。

8.1 调度的目标

本书前面讲过,在多进程、多线程并发的环境里,虽然从概念上看,有多个进程或线程在同时执行,但在单一 CPU 下,实际上在任何时刻只能有一个进程或线程处于执行状态。而其他线程则处于非执行状态。那么这就有一个需要解决的问题:我们是如何确定在任意时刻到底有哪个线程执行,哪些不执行呢?或者说,我们是如何进行线程调度的呢?

进程/线程的调度是操作系统进程管理的一个重要组成部分。其任务是怎么选择下一个要运转的进程。而要探明这一点,则首先需要确定操作系统进程调度的目标是什么。这样,就知道选择什么进程最合适。

那么操作系统进程调度的目标是什么呢?这需要对程序使用 CPU 的模式进行分析。那么程序在执行时有什么样的模式呢?

一般来说,程序使用 CPU 的模式有三种:一种程序大部分时间在 CPU 上执行。另一种程序大部分时间在进行输入输出,还有一种程序则介于前两种模式之间。

第一种程序运行的模式是在 CPU 上执行一阵较长时间,接着进行短暂的输入,然后又在 CPU 上进行较长的运算,之后又进行一下短暂的输入输出操作,就这样循环往复。这种程序由于使用 CPU 的时间大大多于其用于输入输出上的时间,因此称为 CPU 导向(CPU-bound)或计算密集型程序。计算密集型程序通常是科学计算方面的程序。计算宇宙大爆炸各种参数的程序和矩阵乘法程序等就都是 CPU-bound 的程序。

第二种程序则与第一种相反,这种程序的大部分时间用来 I/O,每次 I/O 后进行短暂的 CPU 执行,因此称为 I/O 导向(I/O-bound)或输入输出密集型程序。一般来说,人机交互式程序均属于这类程序。如游戏程序,讲课时使用的 PPT 程序,就都属于 I/O-bound 的程序。

第三种程序自然介于二者之间,既有长时间的 CPU 执行部分,又有长时间的 I/O 部分。或者说,这种程序使用 CPU 和 I/O 的时间相差不大。这种程序称为平衡型程序。例如,网络浏览或下载、网络视频等就属于此类程序。

自然,对于不同性质的程序,调度所要达到的目的也有所不同。例如,对于 I/O-bound 的程序来说,响应时间非常重要,而对于 CPU-bound 的程序来说,则周转时间(turnaround)就很重要,对于平衡型程序来说,则进行某种响应和周转之间的平衡就显得重要。

8.2 处理器调度的总体目标

CPU 调度就是要达到极小化平均响应时间、极大化系统吞吐率、保持系统各个功能部件均处于繁忙状态和提供某种貌似公平的机制。

极小化平均响应时间就是要极小化用户发出命令和看到某种结果之间所花费的时间,即减

少做一件工作平均等待的时间;极大化系统吞吐率就是要在单位时间内完成尽可能多的程序,就是单位时间内能完成的工作数量,即整个系统运行效率高;保持系统各个功能部件繁忙就是要让 CPU 和输入输出设备均处于忙碌状态。因为 CPU 非常昂贵,让其闲置显然是一种浪费,因此保持 CPU 繁忙十分重要。就像人非常珍贵,因此要一直保持学习繁忙状态,才能不浪费生命。

提供公平就是要让各个程序感到某种“平等”,即在 CPU 面前“人人平等”。公平是任何系统都应该努力达到的目标。因为,没有公平,该系统对用户的吸引力就会急剧下降。就像一个国家或者社会,如果缺乏公平,公民对该国家的认同度就会急剧下降一样。

对于不同的系统来说,在调度目标方面也有一些细微的不同。例如,对于批处理系统来说,由于用户并不坐在计算机前面等待结果,响应时间就显得不太重要,但系统吞吐率、CPU 利用率和周转时间则很重要。对于交互式系统来说,因为用户在等待计算机,因此响应时间要很快。但在这里要注意的是适度性(propportionality)。适度性就是响应时间要和期望值相匹配。这里是说你不要超越用户的期望。比如说,用户期待 1 秒钟的响应时间,你就给他 1 秒钟的响应时间,而不必提供 0.1 秒的响应时间。这是因为,提供超出用户期望的响应会增加系统设计的难度,而又不会提高用户的满意度(对于一个人来说,1 秒钟和 0.1 秒钟的差别并不是很大)。对于实时系统来说,调度就是要达到在截止期前完成所应该完成的任务和提供性能可预测性。

8.3 先来先服务调度算法

先来先服务调度算法缩写为 FCFS(first come first serve)。就是谁先来,就先服务谁。这个算法所有地球人都能想到。因为先来先到是人的本性中的一个公平观念,而且生活实际中这种规则随处可见。例如,我们排队买东西或者办理政务体现的就是先来先到原则。

先来先到的一个隐含条件就是不能抢占,一个程序一旦启动就一直运行到结束或者受阻塞为止。这是因为一旦允许抢占,就破坏先来先到的原则了。这类似于我们生活中的干部终身制。先来先到的优点就是简单,人人都能理解,实现起来容易。而缺点则是短的工作有可能变得很慢,因为其前面有很长的工作。这样就造成用户的交互式体验也比较差。

例如,有两个程序:A 需要运行 100 秒,B 需要运行 1 秒。A 程序与 B 程序几乎同时启动,但 B 就是慢了一丁点,被排在 A 之后执行,则需要等 100 秒。这样 A 的响应时间为 100 秒,而 B 的响应时间则为 101 秒,从而,平均响应时间 100.5 秒。响应时间非常慢。

就像我们排队办理事情,你要办理的事情只要几分钟就可办好,而你前面的一个人办理的事情因为复杂需要 1 个小时。这个时候你要等在他后面就十分不高兴。这个时候你就想,要是每个人轮流办理 10 分钟事务的话,那多好呀。

自然,研究处理器调度的人也想到了这点,而这种轮流办理的调度方式就是时间片轮转。

8.4 时间片轮转

时间片轮转算法是对 FIFO 算法的一种改进,其主要目的是改善短程序的响应时间,其方法

就是周期性地进程切换。例如每1秒钟进行一次进程轮换。这样,短程序排在长程序后面也可以很快得到执行。因此长程序执行1秒后就得把CPU让出来。这样整个系统的响应时间就得到改善。以前面的A、B程序为例,A需要运行100秒,B需要运行1秒。使用FIFO时系统平均响应时间为100.5秒。而使用时间片轮转,则A在执行1秒后,CPU切换到进程B,在执行1秒后,B结束,A接着执行99秒。这样A的响应时间是101秒,而B的响应时间为2秒。系统的平均响应时间是51.5秒。显然比FIFO强多了。

仔细的读者可能已经看出,系统响应时间依赖于时间片的选择。我们因为选择了1秒钟的时间片,上述系统的响应时间是51.5秒。如果时间片是10秒钟,则上述系统的平均响应时间将是65秒。如果选择别的时间片,则响应时间还将不同。那我们自然想知道,到底选择多大的时间片才合适呢?

显然,如果时间片选择过大,时间片轮转将越来越像FIFO,当时间片的选择超过任何一个程序所需要的执行时间长度时,则完全退化为FIFO。而时间片如果选择过小,则进程切换所用的系统消耗将太多,使得系统的大部分时间花在进程的上下文切换上,而用来真正执行程序的有用时间很少,从而降低系统效率,并造成浪费。

那如何选择一个合适的时间片呢?做研究。我们需要知道进行一次进程切换所用系统消耗和我们能够承受的整个系统消耗,就可以得出合适的时间片。例如,如果每次进程切换需要消耗0.1毫秒的CPU时间,则选择10毫秒的时间片将浪费约1%的CPU时间在上下文切换上;如果选择5毫秒的时间片,浪费为2%;20毫秒的时间片浪费为0.5%。如果我们能够承受的CPU浪费为1%,则选择10毫秒的时间片就很合理。

时间片选择还需考虑的一个因素是,有多少进程在系统里运行?如果运行的进程多,时间片就需要短一些,不然,用户的交互体验会很差。进程数量少,时间片就可以适当长一些。因此,时间片的选择是一个综合的考虑,需要权衡各方利益,进行适当的折中。

时间片轮转看上去非常公平,响应时间非常好,每个进程周期性的获得CPU时间。但时间片轮转真的很公平吗?时间片轮转的系统响应时间总是比FIFO的响应时间短吗?答案却是不一定。

还用上面的例子,如果是B比A略微先到,如果用FIFO,B的响应时间为1秒,A的响应时间为101秒,这样系统的平均响应时间为50.5秒。而这是最优的响应时间。如果使用时间片轮转,除非时间片选择的是1秒,否则时间片轮转的系统响应时间将比FIFO慢(记住,进程切换是需要消耗系统时间的)。

8.5 短任务优先

诚然,轮流坐庄和先来先到到底哪一种方式更公平是仁者见仁、智者见智的一个话题。但是一个事实是现在的社会都不太认同终身制,而更认同轮流坐庄。18世纪以前,欧洲的皇帝是终身制,逃到美国的清教徒则显然不喜欢终身制,从而选择了4年一轮的总统制度。而这种轮流坐庄的制度已被世界绝大多数国家和社会所认可,皇帝终身制度已被世人唾弃。从这点来说,时间片轮转比起FCFS来似乎更加公平。

那么时间片轮转真的那么公平吗？当然不是。如果每个人的能力一样，大家轮流坐庄当然比较公平，问题是并不是每个人的能力完全一样。让一个能力很差的人与一个能力很好的人轮流执政恐怕没有多少人会同意。我们不是因为不满大锅饭而提出让一部分人先富起来吗？而时间片轮转就是大锅饭！

我们前面举例说了，时间片轮转改善了所谓的系统响应时间的论断也不一定经得起推敲。更为重要的是，时间片轮转所达到的系统响应时间并不是我们所能达到的响应时间下限。如果有 30 个用户，其中一个用户只需要 1 秒钟时间执行，而其他 29 个用户需要 30 秒钟执行，如果因为某种原因，这个只要 1 秒钟的程序排在另外 29 个程序的后面轮转，则需要等待 29 秒钟才能执行（假定时间片为 1 秒）。这个程序的响应时间和交互体验变得非常差。

那要改善短任务排在长任务后面轮转而造成响应时间和交互体验下降的办法就是短任务优先算法 STCF (shorted time to completion first)。这种算法的核心是所有的程序并不都一样，而是有优先级的不同。具体说来，就是短任务的优先级比长任务的高，而我们总是安排优先级高的程序先运转。就像晚辈在公交汽车上遇到长辈时需要让座一样。

短任务优先算法有两个变种：一种是非抢占，一种是抢占。非抢占短任务优先算法的原理是让已经在 CPU 上运行的程序执行到结束或阻塞，然后在所有候选的程序中选择需要执行时间最短的进程来执行。抢占式短任务优先算法则是在每进来一个新的进程就需要对所有进程（包括正在 CPU 上运行的进程）进行检查，谁的时间短，就运行谁。

显然，由于短任务优先总是运行需要执行时间最短的程序，其系统平均响应时间在我们目前已经讨论过的几种调度算法里面是最优的。这就是 STCF 的优点。事实上，在所有非抢占调度算法中，STCF 的响应时间最优。而在所有抢占调度算法中，抢占式 STCF 的响应时间最优。

下面我们来看一个例子：假定有 A、B、C 三个进程，A、B 均是纯计算进程，分别需要使用 CPU 计算 50 和 100 毫秒，而 C 每计算 1 毫秒后进行 9 毫秒的输入输出操作，并这样重复 10 次。

显然，如果 A、B 单独运行，则 CPU 利用率是 100%，如果 C 单独运行，则磁盘利用率为 90%。如果我们将它们一起运行，结果会怎么样呢？这里假定我们使用 STCF 调度算法。

要使用 STCF 算法，首先得搞清楚哪个工作是短工作，哪个是长工作。A、B、C 三个进程相对来说，C 是短工作，因为其使用 CPU 的时间远远小于其花在 I/O 上面的时间。而 A、B 皆是长工作，但相对来说，A 又是短工作，因为其使用 CPU 的时间小于 B 进程。这样，STCF 调度的优先级就是 C、A、B。由此我们获得如图 8-2 所示的调度模式：

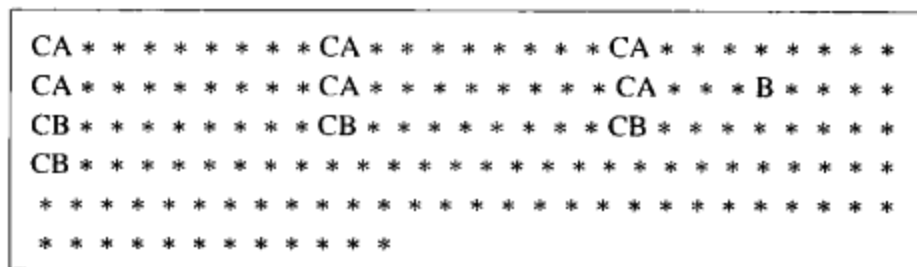


图 8-2 使用 STCF 调度算法对 A、B、C 三个进程的调度情况（每个字符占用 1 毫秒）

在 STCF 调度下，磁盘在 90% 的情况下保持繁忙，这与只运行 C 一个进程的结果相同。A 进程在系统启动后 55 毫秒结束，B 进程在系统启动后 159 毫秒结束，C 进程在系统启动后 99

毫秒结束。故整个系统的平均响应时间为 104.3 毫秒。

如果使用 FCFS 算法，如果 A 或者 B 在 C 之前达到，则磁盘将闲置 150 毫秒后才能第一次启动，磁盘的利用率将大大低于 STCF 调度算法。而系统的响应时间（假定 A 在 B 前面）为 A 为 50 毫秒，B 为 150 毫秒，C 为 250 毫秒，平均响应时间为 150 毫秒。如果 C 在 A、B 前到达，由于 C 在执行 1 毫秒后就阻塞进行 I/O，整个状况与 A、B 先来的时候差不多，这个留待读者完成。

如果使用时间片轮转，假定时间片大小为 10 毫秒，按照 C、A、B 的顺序轮转，我们获得图 8-3 所示的调度情况。

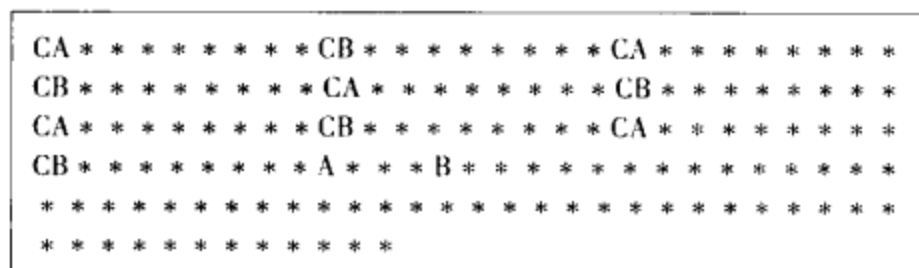


图 8-3 使用时间片轮转调度算法对 A、B、C 三个进程的调度情况

在时间片轮转情况下，系统的响应时间是 A 为 104 毫秒，B 为 159 毫秒，C 为 99 毫秒，平均响应时间为 120.67 毫秒。

由此可见，STCF 的响应时间确实最短。当然，在时间片轮转的情况下，时间片大小选择的不同，结果将有所不同，但其响应时间不会短于 STCF 的响应时间。

但 STCF 调度也有缺点。第一是可能造成长程序无法得到 CPU 时间而导致饥饿。除此之外，还有一个重大缺点，就是我们怎么知道每个进程还需要运转多久？难道我们能够预测将来不成？就好像我讲这个课，我第一次讲时怎么知道会讲多长时间呢？

这个时候就需要做研究！我们可以用一些启发式（heuristic）方法来进行估算，例如，根据程序大小来推测一个程序所需 CPU 执行时间。但这个方法并不可靠。另外一个办法就是先将每个程序运行一遍，记录其所用 CPU 时间，这样在以后的运行中，即可根据这个实测数据来进行 STCF 调度了。

8.6 优先级调度

前面讲过的 STCF 有一个缺点是可能造成长进程饥饿。但这个问题比较容易解决，使用优先级即可。优先级调度算法的原理是给每个进程赋予一个优先级，每次需要进程切换时，找一个优先级最高的进程进行调度。这样，如果我们给长进程一个高优先级，则该进程就不会再有饥饿。事实上，STCF 本身就是一种优先级调度，只不过它给予短进程高优先级而已。

优先级调度的优点是可以赋予重要的进程以高优先级以确保重要任务能够得到 CPU 时间。其缺点则与 STCF 一样，低优先级的进程可能会饥饿。不过，这个问题在优先级调度算法里比在 STCF 里好解决：只要动态的调节优先级即可。例如，我们在一个进程执行特定 CPU 时间后将其优先级降低一个级别，或者将处于等待进程的优先级提高一个级别。这样，一个进程如果等待时间很长，其优先级将因持续提升而超越其他进程的优先级，从而得到 CPU 时间。这样，

饥饿现象就可以防止。

不过，优先级调度还有一个缺点，就是响应时间不能保证，除非将一个进程的优先级设为最高。即使将优先级设为最高，但如果每个人都将自己进程的优先级设为最高，响应时间还是无法保证。

8.7 混合调度算法

我们前面提到过的所有算法都存在缺点，我们自然想设计一个算法合并它们的优点，摒弃它们的缺点。这就是所谓的混合调度算法。该算法的原理是将所有进程分成不同的大类，每个大类为一个优先级。如果两个进程处于不同的大类，则处于高优先级大类的进程优先执行；如果两个进程处于同一个大类，则采用时间片轮转来执行，如图 8-4 所示。

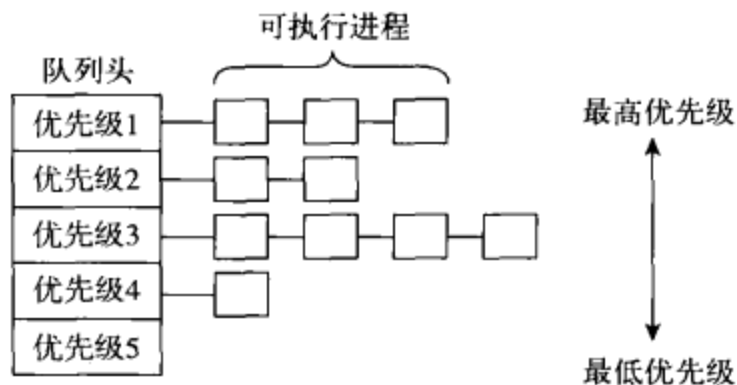


图 8-4 混合调度算法

8.8 其他调度算法

除了上述介绍的各种算法外，有的操作系统还实现了一些其他算法，它们包括：保证调度（Guaranteed scheduling）、彩票调度（Lottery scheduling）、用户公平调度（Fair share scheduling per user）。其中保证调度算法的目标是保证每个进程享用 CPU 的时间完全一样，即如果系统里一共有 n 个进程，则每个进程占用 CPU 的时间为 $1/n$ 。保障调度就是保障每个进程使用 $1/n$ 的 CPU 时间。保障就是肯定 $1/n$ 的时间运转，而不是大概 $1/n$ 时间运转。那么保障调度和轮转调度是一样吗？时间片轮转能不能达到 $1/n$ 的效果？关键就是达到 $1/n$ 不一定要靠轮转。轮转是能够达到 $1/n$ 的，但是保障调度不一定要轮转。每次给的时间片不一定要一样。

彩票调度算法是一种概率调度算法。你买过彩票就知道，你买的越多中奖的概率越大。在该算法里，给每个进程分发一定数量的彩票，而调度器则从所有彩票里随机抽取一张彩票，持有该彩票的进程就获得 CPU。这样，如果想让某个进程获得更多的 CPU 时间，我们可以给该进程多发几张彩票。彩票算法的优越性是显而易见的，通过给每个进程至少一张彩票就可以防止饥饿，因为该进程获得 CPU 的概率将大于 0。除此之外，彩票算法还可以用于模拟其他进程调度算法。例如，如果给每个进程一样多的彩票，则该算法就近似保证调度算法；如果给短任

务赋予更多的彩票，则将类似于短任务优先算法。

那么彩票调度有什么用呢？比如你要保障 A 进程 50% 的时间，那么就把一半的彩票分给 A，这样的话就能保障 50%。别的调度方法也可能达到这个效果，但是不灵活。

用户公平调度算法则按照每个用户，而不是每个进程来进行公平分配。前面讲过的算法均以进程为单位。这样一个贪婪的用户可以通过启动许多进程来夺占 CPU 时间。如果每个用户都很贪婪，都试图启动很多进程，则将造成整个系统效率低下，甚至停顿。用户公平调度算法就是将 CPU 时间按照用户进行平均分配。如果一个用户的进程多，则其所拥有的进程所获得的 CPU 时间将短，反之则多。

自然，CPU 调度算法并不只有上面介绍的几种。由于不同系统的目标不同，不同进程的性质和重要性不同，进程调度也就不同。事实上，调度算法非常多，有兴趣的读者可以参阅在有关调度方面的大量的论文。

8.9 实时调度算法

实时系统是一种必须提供时序可预测性的系统。由于其应用范围广和特性不同于一般计算机系统，其调度算法也别其一格，和我们前面讲过的所有调度算法均有所不同。前面的算法主要考虑的是平均响应时间和系统吞吐率的问题，而实时系统则考虑每个具体任务的响应时间必须符合要求，即每个任务必须在什么时间之前完成，而无需考虑如何降低整个系统的响应时间或吞吐率。

比如计算来袭导弹轨迹的进程，其计算时间是非常有限的。如果进程不能在规定时间内计算出来来袭导弹的轨迹，则结果毫无意义。但如果能够在截止时间前完成，那提前多少则无关紧要。这就是说，只要达到一定响应时间后，再提升响应时间并不能获得任何好处。比如，视频输出，在 NTSC 制式下（美国、日本、台湾使用的电视视频制式），只要每 33 毫秒发出一个图像帧，所看到的视频就是连贯的。而如果发送得比这更快，也不会获得任何额外的好处。其他实时系统还有物理控制系统，如核反应堆控制温度的系统、汽车测速机制等。

实时系统调度算法种类繁多，本书不可能覆盖。这样我们只论述一下其最主要或者说最经典的两种算法：动态优先级调度和静态优先级调度。动态优先级调度又称为最早截止任务优先算法（EDF, earliest deadline first），而静态优先级调度又称为最短周期优先算法（RMS, rate monotonic scheduling）。

8.9.1 EDF 调度算法

该算法就是最早截止的任务先做。如果新的工作来了，比正在运行的程序的截止期更靠前，那么就抢占当前进程。EDF 算法是实时调度里面的最优算法。如果一组任务可以被调度的话（指所有任务的截止时间在理论上能够得到满足），则 EDF 可以满足。一批任务如果不能全部满足，那 EDF 能满足的任务数最多。这就是它最优的体现。

例如，任务 A 需要 15 毫秒执行时间，截止时间在进入到系统后的第 20 毫秒，B 需要执行 10

控制，从而超越这两个任务。

在某些时候，优先级倒挂并不会造成损害。高优先级任务的延迟并不会被注意。因为低优先级进程最终会释放资源。但在其他一些时候，优先级倒挂则可能引起严重后果。如果一个高优先级进程一直不能获得资源，有可能造成系统故障，或激发事先定义的纠正措施，如系统复位。例如，美国的火星探测器 Mars Pathfinder 就是因为优先级倒挂而出现故障。

如果高优先级进程在等待资源时不是阻塞等待，而是循环（繁忙）等待，则将永远无法获得所需资源。因为此时的低优先级进程无法与高优先级进程争夺 CPU，从而无法执行，进而无法释放资源。这将造成高优先级进程无法获得资源而继续推进。

优先级倒挂还可能造成系统性能降低。低优先级进程之所以优先级低是因为其所执行的任务并不重要。例如，它们可能是批处理任务或其他非交互式任务。而高优先级任务执行的则是较为重要的任务，如为交互用户提高数据或实时任务。由于优先级倒挂造成低优先级任务在高优先级任务之前执行，系统的响应将降低，甚至实时系统的响应时间保证都有可能受到违反。

倒挂的解决方案

优先级倒挂的问题在上世纪 70 年代就已经发现，但却没有找到一个可以预测其发生的方法。不过，虽然我们不能预测其发生，但却可以采取手段防止其出现。到目前为止，解决优先级倒挂的办法可以归结为如下几种：

- 使用中断禁止

这种办法的核心是通过禁止中断来保护临界区。在采用此种策略的系统中只有两个优先级：可抢占优先级和中断禁止优先级。前者为一般进程运行时的优先级，后者为运行于临界区的进程的优先级。由于不存在第三种优先级，优先级倒挂无法发生。由于系统里只存在一个锁（中断禁止操作在任何时候只能由一个进程执行），乱序不会发生，因此死锁也不会发生。又由于临界区总能够不被打断而一直运行到结束，悬挂（hang）也不会发生。

这里需要注意的是所有中断都必须禁止。如果禁止的仅仅是一个特定的硬件设备的中断，则硬件的中断优先处理机制将再次引入优先级倒挂。

在多 CPU 环境则使用一个简单的变种：单一共享标志锁。该方法在共享内存里面提供一个单一标志。所有 CPU 进入跨 CPU 临界区时都必须先获得该标志。由于 CPU 间的通信昂贵并且慢，大多数此种系统都尽量不共享资源。因此这种方法在多数实际系统里都效果良好。

该方法普遍应用于简单的嵌入式系统。这种系统的特点是可靠性、简易性和资源需求低。不过这种方法对程序员的要求较高，因此在程序设计时需要将临界区设计的很短，通常应该在 100 微秒以下。而这个时间对于通用计算机来说很不现实。

- 优先级上限（priority ceiling）

在此种方式下，共享的 mutex 进程（操作系统代码）有其自身的高优先级。一个程序如果进入 mutex 保护的临界区，将获得该临界区所具有的高优先级别。此时如果其他试图访问 mutex 的进程的优先级都低于 mutex 的优先级，则优先级倒挂将不会发生。

- 优先级继承（Priority inheritance）

了欧洲大陆。

显然，人们对事情的取向是基于人们对事务状态的判断。不同的判断自然导致不同的行为。在文件实现上也一样，不同的看法和判断导致了文件实现的不同机制的出现。

前面一章讲了从用户角度看到的文件系统，也就是文件系统的感性知识。但对于操作系统设计人员来说，单有感性知识是不够的。我们需要的是文件系统的实现细节，因为这就是操作系统设计人员所看到的层次。这一章从操作系统设计人员角度分析文件系统，也就是文件系统的实现。对于操作系统设计人员来说，他们关心的问题是：

- 文件系统是如何分布的。
- 文件是怎么实现的。
- 文件夹是怎么实现的。
- 共享文件是怎么实现的。
- 磁盘空间是如何管理的。

前面我们讲过，文件主要在磁盘上实现。当然你也可以将文件系统放到其他介质上。但不管是什么介质，其原理大同小异。这里以磁盘为介质进行说明。

买过磁盘的人都知道，磁盘买来后要做的第一件事情是对磁盘进行分区和格式化。那么磁盘为什么要进行分区呢？或者说我们必须分区吗？

分区的理由多种多样。首先，分区可以方便我们对磁盘的使用，因为不同的分区可以建立不同的文件系统；其次，分区有安全性上的优势。因为一个分区毁坏了，另一个分区仍然可以使用；再次，分区还有可靠性上的优势，因为一个分区的故障不影响另一个分区的运行。这些分区理由虽然是好的理由，但都不是必须分区的理由。

必须分区的理由是对磁盘空间的使用。计算机的内存字长度通常有限，而磁盘地址需要存放在内存字里面。这样，操作系统能够访问的磁盘地址数量就是一个有限数。这个有限数将限制操作系统能够访问的磁盘空间大小。如果一个磁盘容量超过这个上限，多余的空间将无法被访问。例如，假如内存字的长度为 16 位，这样操作系统能够表示的磁盘地址数为 2^{16} ，也就是 65536 个磁盘地址。假定磁盘数据块（每个地址）的大小为 512 个字节，则操作系统能够访问的最大磁盘空间为 33 554 432 个字节，即大约 32MB 的空间。

当然，我们可以加大磁盘数据块的尺寸使得能够访问的磁盘空间增大。但这样带来的坏处是磁盘空间的浪费，因为很多文件占不到一个磁盘块。再说，即使这样，也只不过增大能访问的磁盘空间，并不能从根本上解决问题。但如果使用分区，则可以将磁盘分解为大小为 32MB 的多个分区，这样整个磁盘就都可以被访问到。

那么为什么一个分区只能建立一个文件系统呢？

17.1 文件系统的布局

一个磁盘分解为一个个扇面，编号从 0 递增，整数计数。第 0 个扇面在整个文件系统中占

有重要意义。该扇面存放的是主引导记录 (Master Boot Record, MBR)。该记录的内容是一个小程序, 用来启动计算机。如果该扇面损坏, 则整个磁盘就无法使用。这时唯一的办法就是将磁盘拿到硬件制造商那里对磁盘扇面重新编号。

在 MBR 后面紧接着的是磁盘分区表。磁盘分区表里给出的是磁盘的所有分区及其开始地址和终结地址。其中的一个分区为主分区。操作系统就装载在这个分区里。主分区里面最前面的是引导记录 (BOOT RECORD)。

在计算机启动时, 处于主板 ROM 里面的 BIOS 程序首先运行。BIOS 在进行一些基本的系统配置扫描后对磁盘的扇面 0 进行读操作, 将 MBR 里面的程序读到内存并运行。MBR 程序接下来找到系统主分区, 并将主分区里面的 BOOT RECORD 加载并运行。BOOT RECORD 里面内容是一个小程序, 该程序将负责找到操作系统映像, 并加载到内存, 从而启动操作系统。所有的文件系统都必须按照这种格式存放, 操作系统才能正常启动。

在 BOOT RECORD 记录块后面的内容就因情况而异了。一般来说, 在该记录块后面的磁盘内容布局因文件系统的不同可以不同。但在通常情况下, 紧接着引导记录后面的是一个超级数据块 (SUPER BLOCK), 该块里面存放的是关于该文件系统的各种参数, 如文件系统类型、文件系统数据块尺寸等。在 super block 后面则是磁盘的自由空间, 其后面是 I-NODE 区, 再后面是文件系统根目录区。分区的最后存放的是用户文件和文件夹区。

文件系统的布局如图 17-2 所示。

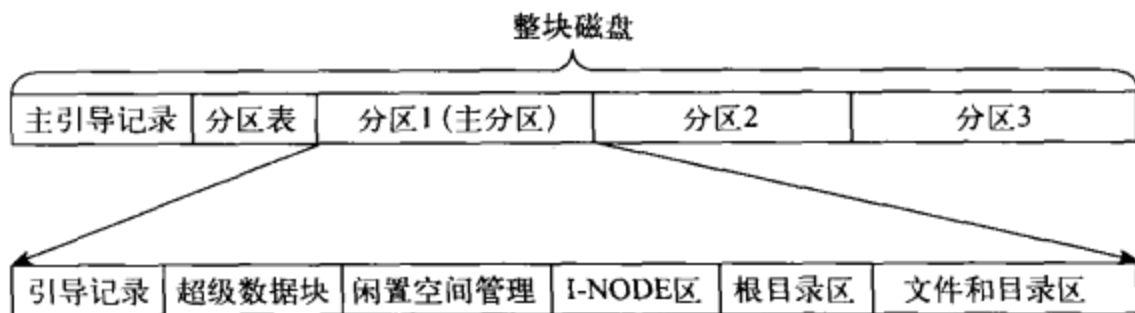


图 17-2 文件系统布局 (来源:《现代操作系统》)

这里需要提醒读者的是, 上述文件系统布局仅仅是一个举例。不同的文件系统布局是不一样的。例如, 对于 FAT 系列的文件系统来说, I-NODE 区是不存在的。因此, 在讨论具体文件系统时, 读者需要参阅该文件系统的手册。

17.2 文件的实现

我们说过, 对于操作系统设计人员来说, 我们关心的是文件如何实现的。那么文件的实现意味着什么呢? 或者文件的实现究竟是什么意思呢?

文件的实现, 归根结底, 就是要能够把文件的内容存放在合适的地方, 并能够在需要时很容易地读出这些数据。这样, 文件的实现要解决的就是如下几个问题:

- 给文件分配磁盘空间。
- 记录这些磁盘空间的位置。

- 将文件内容存放在这些空间。

给文件分配磁盘空间就是要按照用户要求或文件大小分配恰当容量的磁盘空间；记录这些空间的位置对将来的文件访问至关重要；将文件内容存放到这些空间里可以通过磁盘本身的驱动器实现。上述三点均需要了解数据在磁盘上的存放方式是什么。

数据存放的方式，就像程序在内存存放的方式那样，有下面两种：

- 连续空间存放。
- 非连续空间存放。

其中非连续空间存放又可以分为链表方式和索引方式。

17.2.1 连续存放方式

这是任何人都能想出来的办法。在一个文件需要磁盘空间时，给其分配一片连续的磁盘空间，这样一个文件的数据紧密相连，读写起来非常方便。在这种模式下，一个文件占据一片连续的数据块，而这种连续的块通常叫做盘区（extent）。连续存放方式的读写效率很高，因为一次磁盘寻道就可以读出整个文件。数据库文件就要求连续存放。所以装数据库的话最好在一个空的磁盘上装，如果装在一个放了很多东西的磁盘上你就会发现系统运行很慢。

使用连续存放的方式必须要知道一个文件所需要的空间大小。所以使用这种方式必须事先声明所需的最大空间是多少。文件系统则按照这个声明在磁盘上寻找一片足够大的自由空间并把它分配给该程序。而这种寻找自由空间的任务通常由一个适配算法来负责。图 17-3 描述的就是连续存放方式下文件 A、B、C、D、E 的一种可能分配方式。

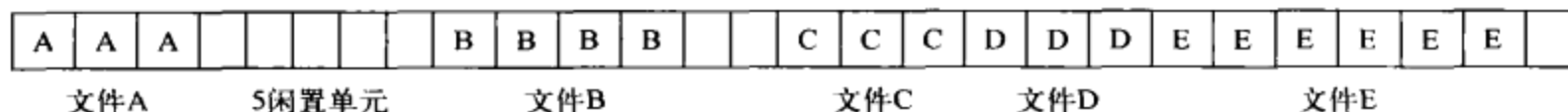


图 17-3 连续存放的文件实现

文件的连续存放方式与程序在内存的连续存放方式非常相似。因此其优缺点也非常相似。具体来说，文件的连续存放的优点就是简单：从文件名到文件地址的映射变为从文件名到文件第一个数据块的地址的映射。即只要给出文件头地址和文件大小，就可以寻找到所有的文件数据块。这个和内存管理中的基址和极限很相似。另外一个优点我们前面已经说过，读写效率很高。

文件的连续存放的缺点自然也与内存的连续分配的缺点一样：空间浪费（磁盘碎片）和不易扩展。在图 17-3 中，如果文件 d 被删除，磁盘上就留下一块空缺。这时，如果新来的文件小于其中的一个空缺，我们就可以将其放在相应空缺里。但如果该文件的大小大于所有的空缺，但却小于空缺大小之和，则虽然磁盘上有足够的空间，但该文件还是不能存放，如图 17-4 所示。

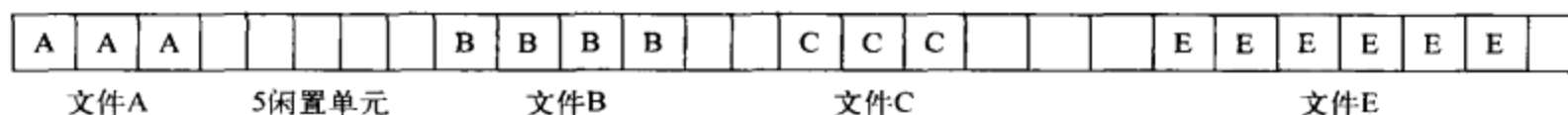


图 17-4 连续存放的文件实现

当然我们可以通过将现有文件进行挪动来腾出空间以容纳新的文件。但在磁盘上挪动文件非常费时，恐怕令人无法忍受。

除了磁盘碎片（空间浪费）外，另一个缺点是文件扩展十分不便。比方说图 17-3 中的文件 D 想扩大一下，需要更多的磁盘空间，有什么办法吗？自然没有什么好办法。唯一的可能是将文件 D 放到磁盘上另一个空间更大的连续片里，或者将与 D 相邻的文件挪开为 D 让出空间。不管哪种办法，效率都非常低。

那有没有什么比较好的办法解决上述问题呢？

有，就是改变文件的存放方式。因为这些文件均是因为文件块连续才造成。因此，我们的解决办法就是非连续存放方式。

17.2.2 非连续存放方式

非连续存放方式就是一个文件的数据块之间不需要在磁盘上占据连续的一片空间。这样，我们需要一种办法来寻找到文件的所有数据块。那么有什么办法可以办到这点呢？答案是链表。

使用链表的办法很简单。在每个数据块里面留出一个指针的空间，用来存放下一个数据块所在的地址。这样一个数据块连着一个数据块，从最前面的数据块开始就可以顺着指针找到所有的数据块，如图 17-5 所示。

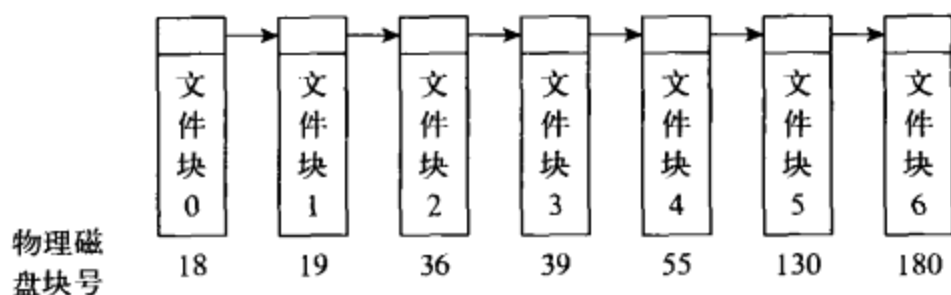


图 17-5 某文件的链表存放方式

在链表存放方式下，文件的映射就是给出文件第一个磁盘块的位置就行了。这种存放方式比起连续存放方式的优点是可以利用碎片。但缺点呢？访问速度慢。尤其是随机访问，访问任何一个数据块均需要从头数据块开始一个指针一个指针地找下去。如果有任何一个指针损坏，则将无法重构整个文件（但如果是连续存放则不存在这个问题，或者说重构要容易得多）。

此种方式的另一个问题是增加了存储开销，因为指针要占空间。那除此之外还有什么缺点？这个缺点你可能得费点力气才能看出来。大家都知道，计算机里面的尺寸都与 2 的指数次方有关系，因为计算机里 2 的整数次方比较容易处理（方便读写）。那么一个磁盘块是多少个字节呢？当然应该有 2 的整数次方个。如果使用磁盘块里面的一部分空间来存放指针，那一个数据块里面存放的数据就很有可能不是 2 的指数次方了。这将造成数据处理的效率下降。

那么我们有什么办法克服上述问题呢？

答案是肯定的。假定我们是文件系统设计人员，我们有什么办法可想呢？把所有指针从单个数据块抽取出来，全部放在一起，形成一张表不就解决问题了吗？这样，要想知道一个数据块的位置，只需要查找该表即可。而且，该表可以存放在内存里面。这样既解决了数据块里面数据不是 2 的指数次方的问题，又解决了随机访问速度很慢的问题。

这张存放文件数据块指针的表我们称为文件分配表 (File Allocation Table, FAT), 如图 17-6 所示。

文件分配表的每个记录为物理磁盘块编号, 每个记录存放的是下一个数据块所存放的物理磁盘块编号。例如, 如果文件 A 的第一个数据块存放在物理磁盘块 3 上面, 那么 FAT 里面索引为 3 的记录里面存放的就是 A 的第 2 个数据块存放的物理磁盘块, 在图 17-6 里面是 10, 即 A 的第 2 个数据块存放在物理磁盘块 10 里。FAT 里索引为 10 的记录的内容是 18, 说明 A 的第 3 个数据块存放在物理磁盘块 18 里。FAT 里索引为 18 的记录内容是 13, 说明文件 A 的下一个数据块在物理磁盘块 13 里, FAT 里索引 13 的内容为 6, 说明下一个数据块在第 6 个物理数据块里。FAT 里索引为 6 的内容是 15, 说明下一个数据块在第 15 个物理数据块里。FAT 里索引为 15 的记录内容为 -1, 说明该文件没有下一个数据块, 即文件已经结束。这样, 根据 FAT 表, 我们得出文件 A 所占的物理磁盘块顺为:

3→10→18→13→6→15

这样, 我们只需要记住文件 A 的起始物理磁盘块为 3 即可。剩下的数据块均可以根据 FAT 表获得。那么我们怎么知道 A 的起始物理磁盘块为 3 呢? 还记得文件夹吗? 这个起始地址存放在文件夹里面。

我们下面看一下 FAT 环境下的随机访问。如果我们要随机读写文件, 则仍然需要从第 1 个数据块地址开始, 在 FAT 表里面顺着指针找到特定数据块所存放的物理磁盘块后才能读取该块数据。但这里的指针跟踪与使用链表时有着巨大的不同: 这里的跟踪在内存发生, 不是在磁盘上发生, 因此效率大大提高。读写任何一个数据块均只需要一次磁盘访问。

这里要强调的一点是, 无论是文件分配表还是链表组织, 只是形式不同而已, 所有链接指针都集中存放, 这和我们下面要讲到的索引文件组织是有本质不同的。

17.2.3 FAT 文件系统

使用 FAT 机制的文件系统就称为 FAT 文件系统。而 FAT 文件系统又有 FAT12、FAT16、FAT32 三种。这三种的区别在于用来表示磁盘地址的内存字位数。如果用 12 位来表示磁盘地址, 则是 FAT12, 用 16 位就是 FAT16。不过 FAT32 却并不是使用 32 位内存字位来表示磁盘地址, 而是 28 位。但因为 28 听上去不像 32 那样像 2 的指数次方, 所以就叫它 FAT32。

FAT 的三种文件系统在 Windows 上均获得了广泛的应用。当然也有些人认为过时了, 不实用了。因为 FAT 表占有内存空间太大。

那么 FAT 表是否占用大量的内存空间呢? 这得看磁盘有多大了。因为 FAT 表的大小与物理磁盘大小和磁盘数据块大小有关。由于 FAT 表的记录项与物理磁盘块数一样, 磁盘越大,

物理磁 盘块号		
0		
1		
2		
3	10	← 文件A起始地址
4		
5		
6	15	
7		
8		
9		
10	18	
11		
12		
13	6	
14		
15	-1	
16		
17		
18	13	
19		

图 17-6 文件分配表

磁盘数据块越小，FAT 表越大。我们在内存分页系统讲过，页表很大是一个大问题，那么这里 FAT 表很大当然也是一个大问题。当然，我们可以采取分页系统里采取的办法，例如通过只将 FAT 表的一部分存放在物理内存，其他存放在磁盘上来减轻这个问题。但这种解决方案的代价则有可能造成文件访问效率的降低。

那还有什么办法改进呢？

当然有。答案就是索引文件组织。

17.2.4 索引文件组织

要改进 FAT 系统，就要看 FAT 的问题出在什么地方。问题就是 FAT 表太大。但仔细分析却发现，FAT 表虽然很大，但里面存放有用信息的记录不一定很多。例如，如果系统里面文件数量较少，或者个体文件的尺寸很小，则 FAT 表里面的很多记录都是空的。这样将整个 FAT 表放在内存里就显得有点不必要了。

那么我们想，如果能够将每个文件的所有数据块的磁盘地址收集起来，集中放在一个索引数据块里，而在文件打开时将该数据块加载到内存，那么以后访问任何一个数据块都可以从该索引数据块里面获得物理磁盘地址。这样，内存里面存放的只是我们需要使用的文件的数据块的所有地址，而不是各种不相干的文件的地址全部一次加载到内存。这样，FAT 占用内存太多的问题就解决了。

这种索引数据块就称为 I-NODE。这里的 I 即是索引（Index）一词的缩写。当然，也有人认为 I 不是代表索引，而是代表信息（Information）。因为 I-NODE 里面存放的不仅仅是文件地址索引，还有其他关于文件的信息，如文件属性等。图 17-7 描述的就是一个 I-NODE。

文件的逻辑数据块编号	对应的物理磁盘数据块编号
0	124
1	354
2	678
3	679
4	800

图 17-7 文件头 I-NODE 举例

使用索引组织方式时，用户先宣称文件大概有多大，文件系统就按照用户的声

明分配一个含有相应数量指针空间的文件头，但是先不分配真正的磁盘空间。在真正需要分配数据磁盘空间时，每分配一个磁盘数据块，文件头里面的相应指针都需要更新。

如果要访问一个文件，则首先将其对应的文件头打开。根据文件头的内容，我们就可以找到任何要访问的数据块。比方说，在图 17-7 里，如果我们要访问第 3 个数据块，则从该文件头里面我们找到 3 所对应的物理磁盘块编号 679，再发出读第 679 磁盘块的命令即可。

细心的读者可能已经发现，这个索引块非常类似于内存管理时的页表。页表存放的是虚拟页面到物理页面的对应，而文件头存放的是逻辑数据块到物理数据块的对应。

既然索引块与页表相似，那页表除了存放地址映射外，还存放一些别的信息。索引块自然也可以在地址对应之外存放别的信息。这正是很多人认为 I-NODE 里面的 I 代表信息的原因。这些信息包括诸如文件创立时间、修改时间、是否系统文件等。

索引文件由于也属于非连续组织方式，自然继承了链表和 FAT 方式的优点。而且，由于

索引块按照个体文件而被加载、打开、关闭，避免了将所有文件的地址同时加载到内存而造成内存空间紧张的问题。另外，随机访问很方便，因为只要从文件头获得所需数据块的物理磁盘地址，一次磁盘访问就可以搞定。索引组织的另一个优点是文件增长灵活。只要文件的尺寸不超过索引头里面预留的指针数 \times 磁盘数据块大小，则文件可以自由地增长。只要在磁盘上找到一个自由磁盘块，分配给文件，再更新文件头以反映这个新的映射即可。

索引组织的缺点也与 FAT 方式类似：比较麻烦。因为每个数据块都需要进行选点，相当于一系列随即访问，即顺序访问效率不太高。

另外，虽然文件增长灵活，但如果一次文件大小的增长超过了预期，其需要的数据块数超过索引头预留的指针数时怎么办？例如，如果只分配了 50 个指针，文件尺寸在 25600 个字节（假定数据块大小为 512 字节）范围内时增长很容易；但如果超过这个范围就不容易增长了。我们当然可以重新分配一个更大的 I-NODE，将原来文件头里面的数据复制到新的文件头，然后再分配新的磁盘空间，更新文件头。

不过这种做法有两个问题：一是需要进行数据的腾挪（从一个文件头挪到另一个文件头），浪费时间；二是使得文件头大小可以任意变化，而这将令文件系统的管理变得复杂。

那么有没有什么较好的办法来解决这个问题呢？

17.2.5 多级索引组织

解决的办法有：多级索引组织。在分页系统里面，页表太大时我们使用多级页表。这里我们自然可以在文件太大时使用多级索引头，即使用间接 I-NODE。

在这种模式下，一个文件使用的 I-NODE 数不是一个，而是多个。而这多个 I-NODE 又被安排成多级：顶级 I-NODE、次级 I-NODE 等。其中顶级 I-NODE 存放的不是数据块对应的磁盘块地址，而是次级 I-NODE 的磁盘地址。次级 I-NODE 里面存放的才是数据块对应的物理磁盘地址，如图 17-8 所示。

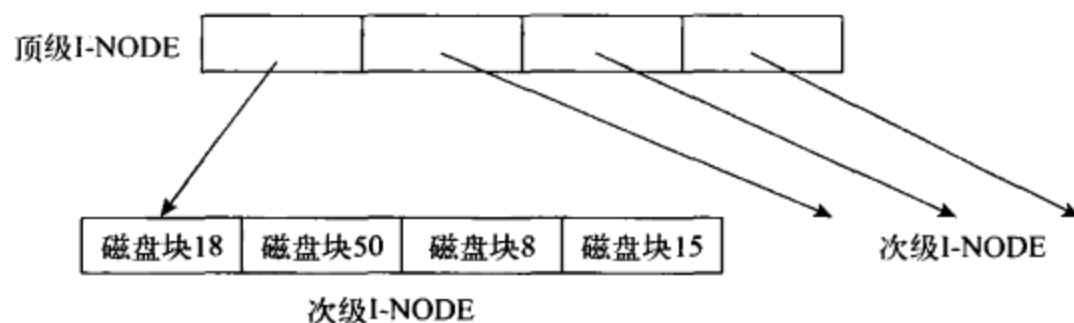


图 17-8 多级索引组织

不过，这种多级索引组织有一个缺点，就是访问数据块所需要的磁盘访问次数增加了。因为访问次级 I-NODE 需要一次磁盘访问，因此磁盘访问速度很低。这种次数的增加显然不好。另外，多级索引还有一个缺点，就是对于小文件来说，它们使用的磁盘数据块可能不会超过一个 I-NODE 里面的指针数。但由于使用多级索引，就白白浪费了 I-NODE 空间。

那么有什么办法解决这个问题呢？有，非对称多级索引。

17.2.6 非对称多级索引

既然单级索引不能适用于大文件，而多级索引又不适用于小文件，那能想到的一种办法自然是单级和多级的有机组合：非对称多级索引。

在非对称多级索引组织下，索引既可以是单级，也可以是多级。到底是单级还是多级取决于文件的大小。所有的文件都有一个顶级索引节点（I-NODE）。不过这里比较特别的是该 NODE 里面的指针域被分为两个部分：一部分用来存放数据块所在的磁盘地址，一部分用来存放次级 I-NODE 的地址。这样，如果文件尺寸较小，其所占数据块数没有超过 I-NODE 里面分配的数据块指针数，则其数据块磁盘地址全部存放在顶级 I-NODE 上。这样的文件将没有次级索引。如果文件较大，则除了顶级数据块磁盘地址外，还将有次级 I-NODE，如图 17-9 所示。

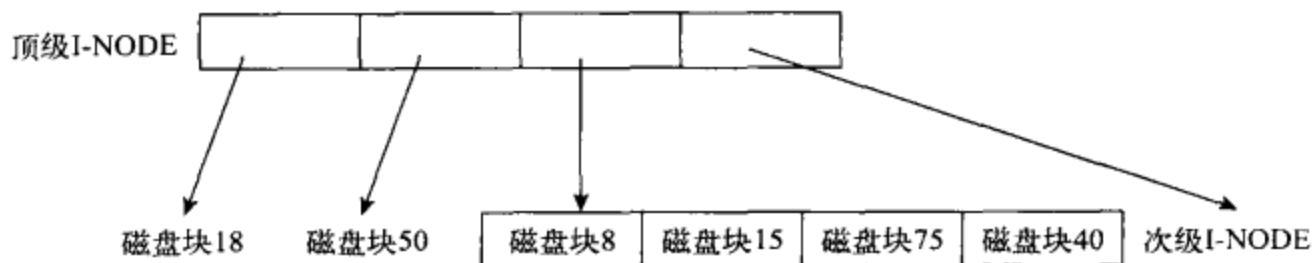


图 17-9 非对称多级索引组织

例如，如果一个 I-NODE 里面可以存放 50 个指针，50 个指针不是全部用来存储数据块对应的磁盘块地址，而是留两个指向磁盘块，称为间接磁盘块。而间接磁盘块上存放的则是数据磁盘块的地址。I-NODE 里面指向数据块的指针称为直接指针，指向间接磁盘块的指针称为间接指针。

非对称多级索引对小文件来说速度块，空间省，对大文件来说可以容纳。但是它对大文件的访问速度提高了吗？从表面上看，没有。因为大文件需要使用多级 I-NODE，进而需要多次磁盘访问。这和普通多级索引没有什么区别。

但真的是这样吗？这个时候如果我们懂一点心理学就会清楚多了。由于使用非对称多级索引，大文件的前面数个数据块的磁盘地址存放在顶级 I-NODE 里，后面的数据块的磁盘地址存放在次级 I-NODE 里。这样，访问前面的数据块显然比访问后面的数据块要快，其速度和单级索引一样。如果我们对这个特点加以利用，先读出前面的数个数据块并呈献给用户，在用户忙于处理前面的数据块时，再在后台读出后面的数据块。这样用户看完或处理完前面几个数据块，需要看到后面时，后面的也读出来了。这样用户的感受是速度很快（像单级索引那样快）。因此，从用户体验的角度看，非对称索引也提高了大文件的访问速度。

那么我们把指针域如何在顶级和次级之间进行分配呢？即多少指针域用来存放直接磁盘地址，多少用来存放间接磁盘地址呢？这就要对文件系统的状况和人的心理进行研究。即研究两个问题，一个是用户的文件一般多大，一个是用户的忍耐能有多久。如果大多数文件都小，那就要多留一点直接块，让用户能够一次读出来。如果文件都大，那就多留间接空间，但却不能将直接空间挤压得过小，使得直接访问的数据量小于用户正常阅读速度 \times 访问次级数据块需要的时间，从而造成用户体验的下降。

如果一级间接还不能容纳系统里的大文件，那就可以使用二级间接、三级间接、四级间接……当然，随着间接层次的增加，文件的访问速度将随着下降，最后可能达到不能容忍的地步。这个凡是打开过巨大文件的人都有过的经历。因此，在实际的商业操作系统里，间接的次数都有个上限。例如，UNIX 的 V7 文件系统的间接层数为 3。

一旦间接层数设定，则一个文件系统里能够存放的文件最大尺寸也就确定了。

17.2.7 文件缓存

单级也好，多级也好，文件的读取相对于内存访问来说就是慢。而对于慢我们有什么解决办法呢？答案是使用缓存。

将文件里面经常要访问的内容存放在缓存里，使得文件的访问可以在缓存满足，而无需到磁盘上去读写。这种缓冲称为文件缓存。

17.3 目录实现：地址独立的实现

我们前面讲过，文件夹的任务是提供从文件名到文件地址的映射。那么对不同的文件组织形式，文件的地址表示也不一样。

如果文件是连续存放的，我们只需要文件的第一个数据块的磁盘地址即可，后面的数据块紧接在该数据块后面。这种情况下文件夹里面存放的映射是到文件头数据块地址。

如果是链表组织形式，我们只需要文件的第一个数据块磁盘地址即可，后面的数据块可以通过前面数据块里面的指针获得。这种情况下文件夹里面存放的映射也是到文件头数据块地址。如果是 FAT 组织形式，映射仍然保持不变。我们可以从 FAT 表里面找到后继数据块所在的物理磁盘地址。

如果使用的是 I-NODE 组织形式，我们只需要知道文件对应的顶级 I-NODE 地址即可。文件的数据块地址可以从 I-NODE 里面获得。这种情况下文件夹里面存放的映射是到 I-NODE 编号。

17.3.1 文件属性的存放

那么文件属性应该存放在什么地方呢？这也是根据文件组织方式的不同而异。

如果是连续或链表组织形式，只有文件夹里面可以存放文件属性。但如果是 I-NODE 组织形式，则属性既可以存放在文件夹里，也可以存放在 I-NODE 里，如图 17-10 和图 17-11 所示。

文件名	文件属性 1	文件属性 2	文件属性 n	起始数据块地址
Mail				180
File	读写				600
Process	只读				

图 17-10 将属性存放在文件夹里

文件名	I-NODE 编号

文件夹

文件属性类别	文件属性值
文件属性 1	
文件属性 2	
.....	
文件属性 n	
文件的逻辑数据块编号	对应的物理磁盘数据块编号
0	124
1	354
2	678
3	679
4	800

图 17-11 将属性存放在 I-NODE 里

17.3.2 长文件名的存放

我们前面说过，文件夹存放从文件名到文件地址的映射。那么每一个文件将在文件夹里面占用一个记录。既然是记录，总有一个长度。为了便于文件的读取，文件夹里面的记录长度通常是固定的。那么对于文件名很长的情况，即文件名长度超过记录能容纳的长度时，我们怎样存放文件名呢？

当然，最简单的办法是不允许长文件名。但这个办法很可能会激怒用户。

另外一个办法则是在目录夹里给长文件名的文件分配多个记录空间，用以存放长文件名。但这样就将造成每个文件占用目录夹记录数不一样的情况。而这将使得文件的读写操作复杂性增加。在这种存放模式下，每个文件夹记录的前面一个字应该存放该文件记录的长度，这样就可以告诉操作系统下一个文件记录开始的位置在哪里。但这种方式要求在查找文件时顺序扫描文件夹，并动态计算下一个文件开始的位置，因此，效率十分低下。图 17-12 描述的是可变长度文件记录的存放方式。

除此之外，还有一个办法是保持文件记录的长度不变，将文件名存放在另一个分开的地方，如系统堆空间（磁盘上专门划分出来的一片空间）。这样，每一个文件记录里面保存一个指针指向堆里存放该文件名的地址。在此种模式下，由于文件记录长度不变，在扫描文件

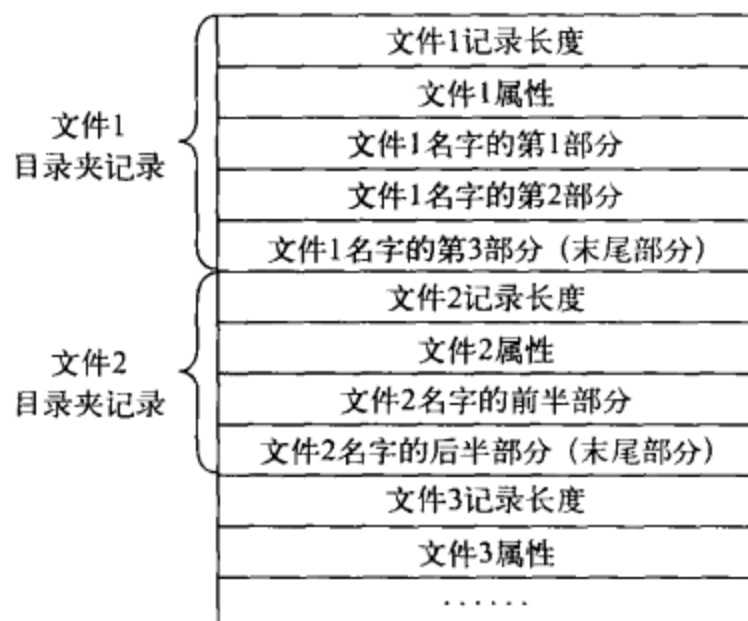


图 17-12 可变长度文件记录方式

夹时无需动态记录下一个文件开始位置，效率将提高。图 17-13 描述的是堆存放方式。

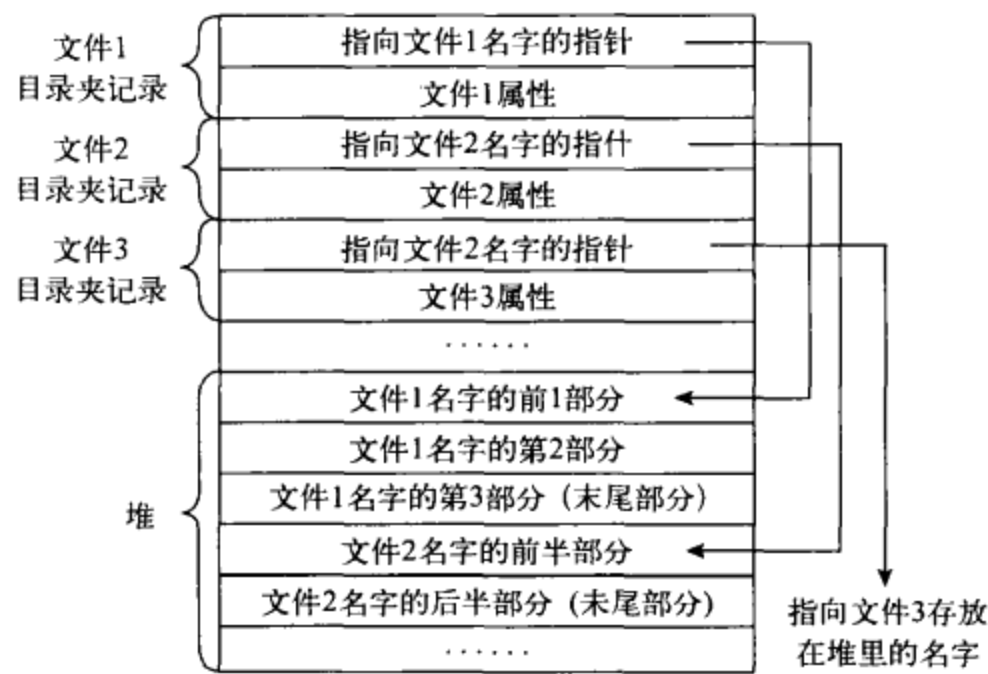


图 17-13 堆存放方式

17.3.3 文件共享

从用户视角看文件时，我们讲过符号链接可以让多个用户共享一个文件，但没有说明链接是如何实现的。现在讲了文件的实现，我们就可以来看一下符号链接是怎么一回事了。

在使用符号链接后，一个文件可以从多个路径进行访问，即一个文件有多个路径名，从多个路径都可以找到该文件在磁盘上存放的地址，也就是说，该文件的映射情况被保存在多个文件夹里。具体来说，就是当用户将一个文件链接到一个文件夹时，该文件夹里面会增加一条记录，用来保存该文件到文件地址的映射。而为了保持系统一致性，我们此时在文件的文件头 I-NODE 里面记录其被链接的次数。图 17-14 描述的是从目录夹 4 链接文件 5 后的情况。

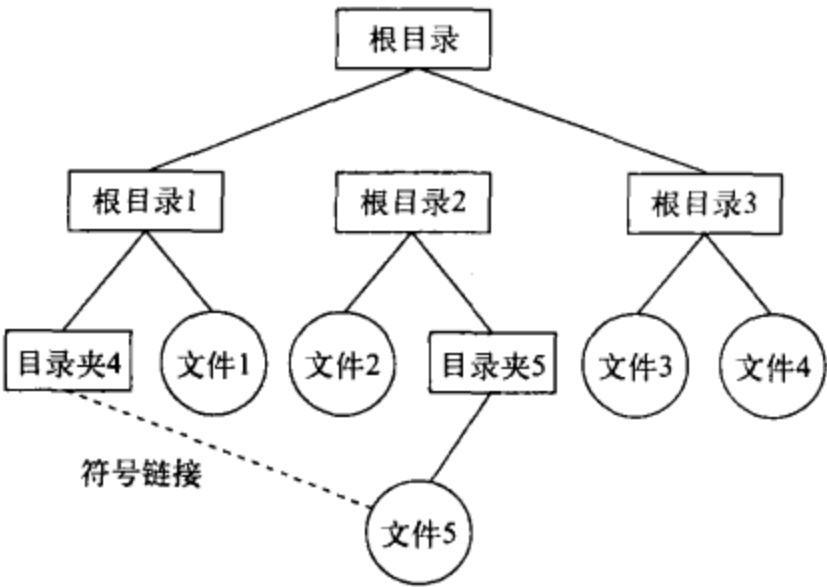


图 17-14 文件链接

有了链接后，文件删除的操作需要进行修改。删除文件时不是马上将文件删除（不管是逻辑上还是物理上），而是将该文件对应 I-NODE 里面的链接计数减一。如果结果为 0，就删除该文件（逻辑删除），如果不为 0，则不删除该文件。这是因为还有目录需要使用该文件。如果这个时候删除了，将出现所谓魅影，即一个文件在目录夹里面显示，但实际却不存在。

17.3.4 硬链接

上面讲到的链接方式将文件的地址映射直接加到链接目录下，这种链接也称为硬链接。这是因为，另外一条路径断开后（如删除文件或者路径上的文件夹），其他的链接不会受到影响。

硬链接有一个大问题。假定 A 用户创立一个文件 `file.pdf`，并且用户 B 使用链接来进行共享。当 B 链接后，A 决定删除该文件。但由于该文件的链接数大于 1，A 的删除操作只是将链接数减一而已，文件并不会被删除。这样 B 用户可以继续使用 A 文件，但 A 文件所造成的成本如占用的磁盘空间成本却要由 A 来负担，因为 A 是其创立者。这显然会令 A 感到不快。而且，这种链接使得文件的创建者没有权利断开用户的链接。

打个比方，一辆车一般有两把钥匙，你将一把钥匙拿给别人保管，如果你把车卖了，那么你是不能控制车了，但是那个保管钥匙的人还能开那辆车，所以这个就是硬链接的问题。

那有没有什么办法既可以提供链接，但又赋予文件创立者更多的控制呢？有，答案是软链接。

17.3.5 软链接

软链接，顾名思义，就是这个链接是软的，经不起折腾。或者说，在创立者将文件删除后，链接用户将无法再访问该文件。那么软链接是怎么实现的呢？

要想将链接变软，就不能将文件的地址直接存放在链接目录的目录夹里。而如果不存放文件的磁盘地址，还有什么办法可以访问到该文件呢？当然是间接了。

就是在链接目录里存放到链接文件的原始路径名（文件被创立时使用的路径名）。即从链接目录访问文件需要走原始路径过来。这样当原始路径断开时，该链接就自然也断开了。这种在原始路径断开后不能维持的链接就称为软链接。

例如，如果从 B 目录建立到 A 目录下文件 `file.pdf` 的链接，则我们在 B 目录下建立一个新的文件，比如说 `file`。而文件 `file` 的内容是 `/A/file.pdf`。

软链接从严格意义上不是链接，因为它并没有直接连到文件上，而是保存了文件的原始访问路径而已。正因为此，Windows 的快捷方式不是链接。因为快捷方式保存的就是路径名。那为什么不用硬链接呢？这是因为 Windows 上经常使用 FAT 文件系统，没有地方存放文件的链接计数，硬链接无法实现。当然，如果是 NTFS，则可以使用链接。但为了保险起见，还是不用为妙。

17.3.6 切断链接

切断链接就是将目录夹里面相应的文件记录删除。对于软链接来说，就是将存放原始访问路径的文件删除。链接切断后，在目录里面将看不到原来被链接的文件。

17.3.7 链接带来的问题

链接虽然提供方便的共享，但也存在诸多问题。最主要的问题是造成备份困难。在进行逻

辑备份时，如果选择备份的文件夹里面存在符号链接，则从逻辑上来看，该文件存在多个路径。这样将造成该文件备份多次。但是备份多次并不是什么大问题，只不过浪费了空间而已。

问题出在恢复的时候。由于被链接的文件备份了多个复本，在恢复的时候自然也会恢复多个复本，这样将造成每个路径有自己的文件复本。这样就达不到共享的目的。当然我们可以对备份软件进行修改，使其能够辨认出符号链接，而只备份和恢复一个复本。但此种修改复杂性较大。

17.3.8 文件系统挂载

知道了文件的实现后，我们就可以理解文件系统是如何实现挂载的了。挂载是用来将一个文件系统并入到另一个文件系统的方法。挂载时需要提供被挂载的文件系统的根目录和挂载点。而挂载实际上就是修改挂载点的目录内容，增加一个记录将该文件系统的根目录 I-NODE 地址保存起来。图 17-15 描述的就是文件系统挂载的情景。

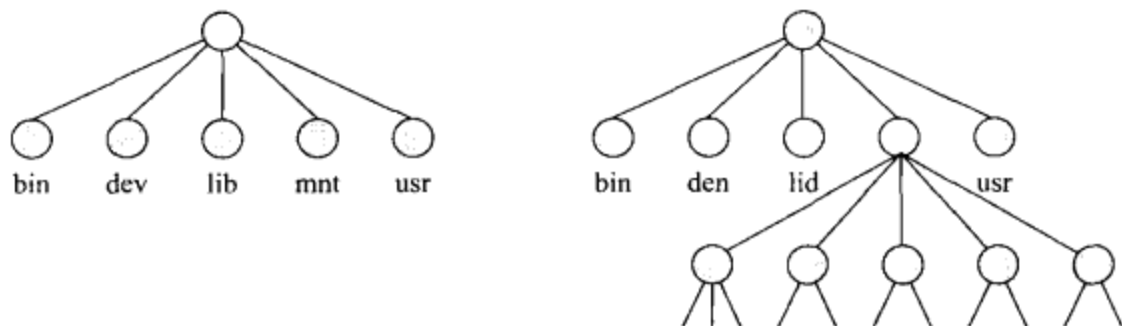


图 17-15 文件系统挂载

比如说，我们平时用的光盘、U 盘、软盘等上面存在一个文件系统。但是该文件系统我们在平时是无法访问的。要想访问，需要将它们插入计算机的相应设备或接口上。而这种插入就是挂载操作，使得光盘、U 盘、软盘上的文件系统成为整个计算机文件系统的一部分，从而我们可以对其进行访问。

在 Windows 操作系统下，挂载点一般是“我的电脑”（My Computer），这是 Windows 里面文件系统的根目录。但在新的 Windows 版本下，也可以挂载在非根目录下。但是在 Windows 下面很少有人将文件系统挂载在其他地方。

在 UNIX 和 Linux，挂载点可以是任何目录。

```
% ls /
afs/      kernel/   sbin/
bin@      lib@     template-server@
core      lost+found/  template-server-ro@
dev/      nfs/     tmp/
devices/  opt/     usr/
etc/      platform/  var/
export/   proc/    vol/
home/     root/
```

17.3.9 卸载

挂上去的文件也可以卸下来，称为卸载。卸载即是将挂载点的目录内容中的相关记录删

除。对于光盘来说，弹出光盘就是卸载光盘文件系统。对于 U 盘来说，拔出就是卸载。

对数据安全十分担心的人可以将文件系统卸下来，从而使得文件系统上的内容无法访问而达到安全的效果。当然最安全的就是根本不用电脑。

17.4 闲置空间管理

文件系统的主要任务是为用户提供一个方便、持久、可靠的存储媒介，并根据用户需要随时对文件的数据进行访问。而为了能够让用户程序进行磁盘空间分配，文件系统必须保持磁盘空间使用情况的记录，即哪些空间被占用，哪些空间为闲置。而对闲置空间进行管理的机制与内存空间管理一样，也是位图和链表。读者可参考内存空间管理，这里不再赘述。

17.4.1 磁盘分配块

文件系统分配空间的时候是按照块（block）来分配的，Windows 操作系统里面将这种分配块称为簇。块或者簇的尺寸在超级数据块中记录着。有了这个参数，我们就可以把字节数转换成块或簇数。

这里有一个问题是：分配块应该为多大？如果分配块尺寸太大了，浪费的空间多，太小了，数据传输速度慢（记得磁盘管理一章的内容吗？）。显然，一个恰当的尺寸是某种折中的产物。而折中的参数就是一个系统里面文件的平均大小。这个分配块大小应当与文件的平均尺寸存在某种联系。图 17-16 给出的是文件平均大小为 2KB 的情况下，磁盘分配块大小与磁盘利用率和有效磁盘数据传输之间的关系。

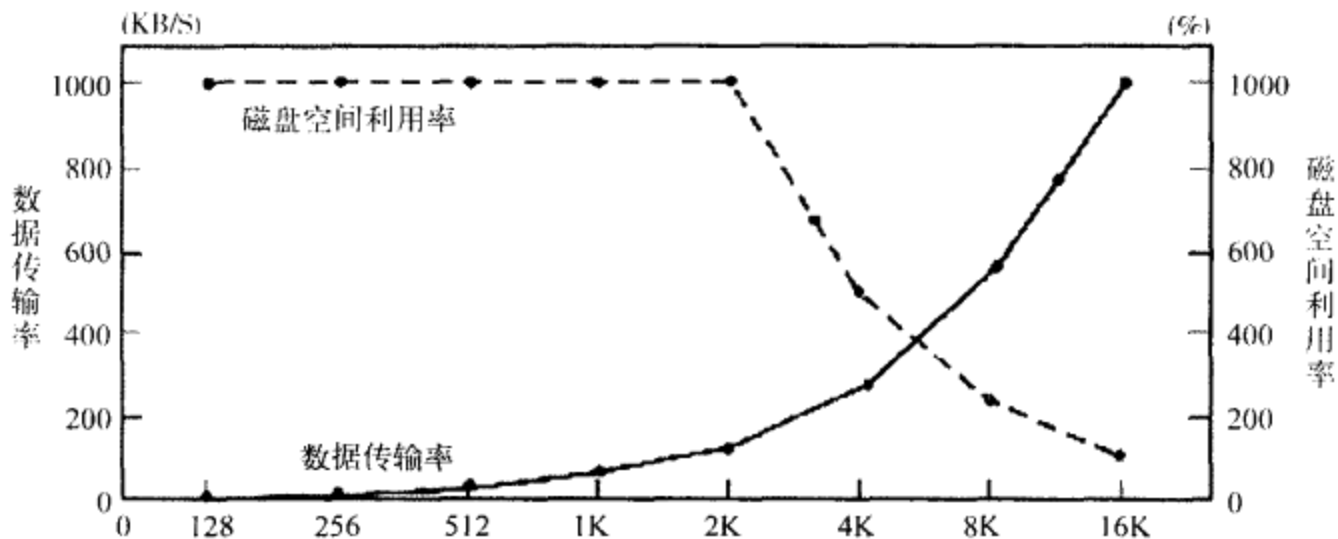


图 17-16 磁盘分配块与磁盘利用率和有效数据传输之间的关系

（来源：《现代操作系统》参考文献 [3]）

17.4.2 磁盘配额

我们大多数人在申请公共邮箱时都会被告知存储空间容量。这个容量就是我们的配额。在多用户环境下，文件系统通常给每个用户一个磁盘配额。这个配额既可以用磁盘容量来表示，

也可以用文件数量来标定。当然,更为常见的是两者同时使用。即一个用户只能创立一定的文件数,且其创立的所有文件加起来不能超过某个容量限度。

对容量配额的实现,不同的操作系统使用的方法不一定相同。在 UNIX 环境下,通常存在所谓的软配额和硬配额。软配额就是你可以临时超过,但必须在下次登录前解决这个问题,否则超出配额的文件将被删除。硬配额就是在任何情况下都不能超过。一旦超过,当前写入的数据都无法完成。磁盘配额的具体实现请参考具体的文件系统,这里不予赘述。

思考题

1. 多级索引的文件系统有何优点? 请予以阐述。
2. 比较连续存放和链表存放方式,哪一种方式的可靠性高?
3. 解决多级索引速度慢的问题可否像解决多级页表那样使用 TLB?
4. 软链接时我们是否需要记录文件被链接的次数? 为什么?
5. FAT 表是为了改善链表结构而提出。比起链表结构,使用 FAT 表将大大加快随机访问的速度。但如果是顺序访问,则 FAT 表似乎并不能提供效率上的提高。请问情况真是这样吗?
6. 有人说: FAT 表在文件系统启动后被调入内存,而磁盘上则仍保留一个复本。那么每次修改内存里面的 FAT 时是否需要修改磁盘上的 FAT 表? 如果修改,有何问题? 如果不修改,又有何问题? 你的策略是什么?
7. 假定我们有一个文件系统,只有 1 个文件。假定磁盘块大小为 1000 字节。而一个用户程序需要进行如下两个操作:
请求 1: 从该文件 8000 偏移量 (offset) 处读一个字节。
请求 2: 从该文件 15 000 偏移量处读一个字节。
该文件有一个文件头 (I-NODE), 而该文件头占一个磁盘块,且不在内存里。请问在下述文件结构下,在不使用缓存的情况下,上述两个请求各需要几次磁盘访问?
a) 顺序、连续分配方式。
b) 非均匀双级索引 I-NODE 结构。I-NODE 里面包含 10 个直接指针 (指向文件的数据块)、2 个间接指针 (指向间接磁盘块)。假定间接磁盘块里面的每个指针占用 4 个字节。
c) 如果一旦数据被读取,就缓存起来,请问答案有何变化?
d) 请问 b) 所支持的最大文件尺寸是多少?
8. 现代 UNIX 支持重命名操作 `rename (char * from, char * to)` 系统调用。该调用将文件名 “from” 重新命名为 “to”。但是 UNIX v6 没有重命名的系统调用,其对重命名的实现是通过链接 `link` 和撤销链接 `unlink` 两个系统调用来实现。
a) 请使用 `link` 和 `unlink` 来实现重命名操作,并阐述你的实现。
b) 如果在执行 `rename (“#x.c”, “x.c”)` 中途计算机崩溃,有可能出现什么情况?
这里假定在该调用前 “#x.c” 和 “x.c” 存在同一个目录里,但在不同的目录块里。
9. 在索引文件系统里,建立新文件有两个操作:写文件数据,写索引块。请问这两个操作应该谁先谁后? 为什么?
10. 在索引文件系统里,写磁盘空间的位图表先于写索引块。请问这是什么原因?

第 18 章 文件系统

引子

耶稣在接受施洗约翰的洗礼后，由圣灵引导，来到旷野接受撒旦的试探。

在节食 40 昼夜后，撒旦（见图 18-1）带耶稣进了圣城，叫他站在殿顶上，然后对他说道“你若是神的儿子，就从这里跳下去。因为经上记着，上帝要吩咐他的天使，将你托在手上，不致使你的脚碰到地上。”

很多人在看到撒旦所说的这段话后，都觉得撒旦的话很有道理，逻辑上无懈可击。因为，如果真是上帝的儿子，上帝自然不会让他从空中摔到地上。就像很多人声称，如果上帝存在，那就在世人面前显现一下，世人不就都信上帝了吗？何必要这么多人费时费力地传播福音呢？这听上去都很有逻辑。

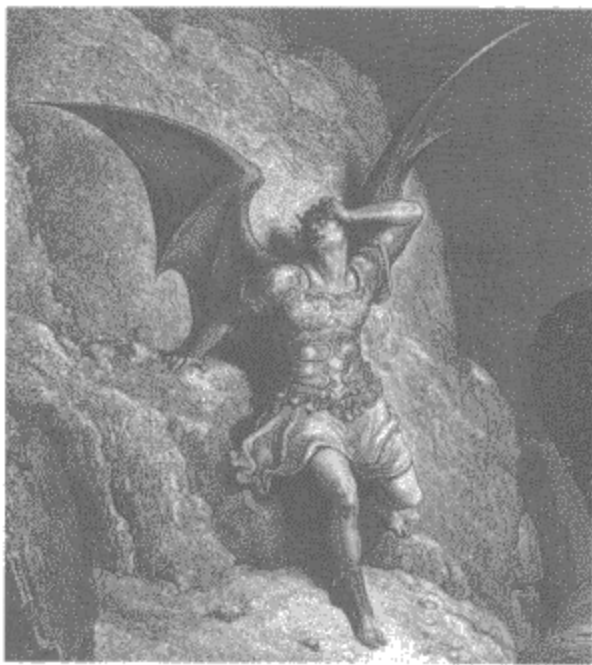


图 18-1 传说中的天使路西佛（lucifer），堕落后成为撒旦

如果一个人觉得撒旦的话逻辑完美，那是因为他没有看到撒旦话里面的巨大漏洞。这个漏洞是什么呢？对于很多人来说，找出这个漏洞似乎是不可能的。

但是，许多乍看上去不可能的事情，其实并不是真的不可能。就像文件系统的演变一样。在文件系统的演变过程中，许多以前被认为是不可能的机制与构造，今天都成了现实。也许，读完本章后，撒旦上述话语里面的逻辑纰漏也就会显现在你的面前。

18.1 文件系统访问控制

前面一章说过，地址独立和地址保护是文件系统需要达到的两个目标。第17章论述了文件系统的第1个目标，即地址独立是如何实现的（地址独立通过文件或文件夹已经得到实现）。现在我们论述文件系统的第2个目标，即地址保护是如何实现的。首先要注意的是，地址保护不是文件系统必须实现的功能。因为如果没有这个功能，用户可以承担起文件保护的角色。例如，用户可以将自己的文件放在一个单独的文件系统上，如U盘或移动硬盘。在工作完成后，将文件系统卸载从而使得自己的文件无法被其他人访问。只不过，此种做法比较麻烦。另外在一些大型主机下也不能实现。因此，由操作系统提供文件保护经常是更为可行的办法，有时甚至是唯一的办法。

那么地址保护如何实现呢？

我们在内存管理时讲过，多道编程的一个重要课题是进程之间的相互保护，即一个进程不能访问另一个进程的空间。也就是说一个进程的数据不能被另一个进程随便访问。这种进程地址空间保护我们说可以通过页表（段表）和动态地址翻译来实现。但是把这个数据存放到磁盘后，动态地址翻译就无法保护这些数据了。那么这些存放在文件里面的数据就面临被任何人或程序访问的危险。而这显然是不能容忍的。如果这个能够容忍，我们何必费力气来保护进程里面的数据呢？至少无需费太大力气来做。

因此，对文件里面的数据进行保护，使得文件数据不能被随意访问就是文件系统必须解决的一个问题。而文件系统解决这个问题的方法就是文件系统的访问控制（Access Control）。那么访问控制是如何实现的呢？或者说，如果要对文件进行保护，使得一个文件不能被任何人随便访问，我们有哪些办法呢？

我们只要看一下人类社会对贵重设施的保护就清楚了。例如，军事重地或关键设施的保护通常可以分为两种模式：在军事禁区设置哨兵。欲访问军事禁区的人必须经过哨兵的检查。即哨兵根据某种规则来确认某个人是否可以访问禁区。另外一种办法是设置安全门。而有权利访问的人皆备有进入安全门的钥匙。如果某个人需要进入禁区，他只需要拿出自己的钥匙将安全门打开即可。如果打不开，则不能访问。在这种情况下，一个人可以有多把钥匙，用来访问不同的受保护设施。

如果对上述情况进行提炼，我们发现，这两类保护措施是从两个不同的角度来实施的，哨兵是从被保护的设施角度出发，即控制措施附在设施上。而使用钥匙则是从用户角度来实施的，即访问控制在用户身上。

既然文件保护的设计者是人，自然我们采取的措施就很难超越我们保护军事禁区的手段。事实上，对文件的保护所采取的措施跟保护军事禁区的手段非常类似。

因此，我们对文件的保护也可以从两个角度出发：从文件的角度和从用户的角度。从文件

的角度实施将访问控制附在每个个体文件上。从用户角度出发则将访问控制附在用户身上。

18.2 主动控制：访问控制表

主动控制将保护措施构建在被保护者身上。对于文件系统来说，我们为每个文件设置一个哨兵。这个哨兵对每个试图访问该文件的用户进行检查，看看其是否能够访问。而这个检查的规则存放在一张表里面。这个表我们称为访问控制表（ACL）。这个表存放在内核空间，对于每个具有访问权限的用户设置一个记录。该记录记载着用户 ID 和具体的访问权限，如读、写、执行、复制等。凡是不在这张表上的用户将不具备访问资格。图 18-2 显示的是有着 3 个用户进程和 3 个文件的访问控制表。

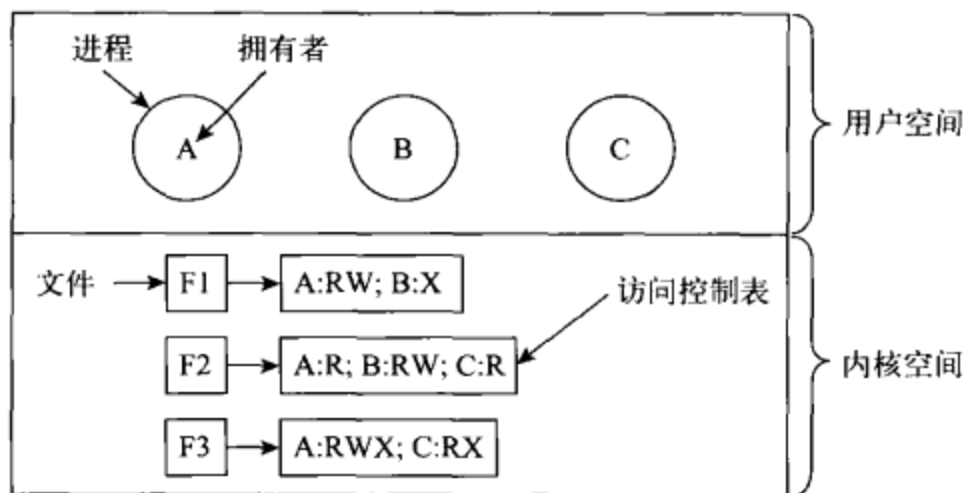


图 18-2 访问控制表

在图中，文件 F1 的访问控制表为：“A:RW; B:X”，这意味着用户 A 可以对文件 F1 进行读写访问，而用户 B 可以对文件 F1 进行执行访问，即可以执行文件 F1。而用户 C 则不能访问文件 F1。对于文件 F2 来说，用户 A 的权限为读，用户 B 的权限为读写，用户 C 的权限为读。对于文件 F3 来说，用户 A 没有权限，用户 B 拥有读写和执行权限，而用户 C 拥有读和执行权限。

这样，在每次访问前，操作系统将检查这个访问控制表来确认一个用户是否有权执行其所要求的操作。如果该用户 ID 在访问控制表里，且其权限与其所要求的操作匹配，则操作将被允许。否则，该操作将被终止。

这里需要注意的是访问控制表是存放在内核空间，由操作系统进行设置与访问。用户不能直接修改访问控制表。

当然，访问控制表里的用户不一定非是个体用户，也可以是集体用户。例如，表 18-1 里面的用户名 faculty 就是一个用户组名，代表所有的教师。任何属于该组的用户都能够读文件 Faculty_Record。Sysadm 也是一个用户组名，代表所有的具有系统管理权限的用户。任何属于该用户组的用户均可以读写 Student_Record 和 Faculty_Record 两个文件。

表 18-1 访问控制权限表

文件	访问控制权限
Student_Record	Zou, sysadm : RW
Faculty_Record	Faculty: R; sysadm : rw

而且，这里的文件也不一定非是个体文件名，而可以是文件集合，如文件夹名。例如，表 18-1 里面的 Faculty_Record 不一定需要是一个用户文件名，它也可以是一个文件夹名。这一点对于读者来说应该不奇怪。我们前面已经说过，文件夹就是文件，可以像对待文件一样对待文件夹。因此，对文件夹设置访问控制权限表就没有任何奇怪之处了。

访问控制表的优缺点

访问控制表的优点是容易理解和实现，对个体用户的权限赋予与取消也容易。只需要将该用户从访问控制表里面删除即可废止该用户对文件的访问，而缺点则是效率不高。每次访问一个文件时，我们需要搜索访问控制表。对于那些不支持用户组的系统来说，将需要给每一位个体用户进行权限赋予，而这将是十分繁琐的。

另外，访问控制表并不十分可靠。它可以很容易地攻破。而攻破的办法也很简单：攻击者只要让系统认为他是某个拥有访问权限的用户即可。例如，我们知道 UNIX 下的 sendmail 程序是在根用户下运行，即它具有根用户的权限。一个攻击者可以对 sendmail 进行颠覆，在其中插入攻击代码。而这段攻击代码将具有根用户的所有权限，因为系统认为该程序就是根用户的程序。

18.3 能力表

另外一种访问控制手段是将控制措施附在用户身上。在这种情况下，每个用户手上拿着一把“钥匙”。这把钥匙就是用户具有的能力。能力 capability 这个术语首先出现在邓尼斯和冯霍尔于 1966 发表的一篇文章：(*Programming Semantics for Multiprogrammed Computations* 《多道编程计算里的程序语义》)。其基本思想是这样的：如果一个计算机程序需要访问一个对象，该程序必须具备一个特殊的令牌。该令牌记录该程序可以进行的操作。而该令牌被称为能力。

由于一个用户可以拥有对于多个对象的访问能力，这些能力放在一起就形成一张表，我们称为能力表。这个表为所有该用户具有访问权限的文件设置一个记录。该记录记载着文件名和具体的访问权限：如读、写、执行、复制等。例如，`< file2 R, file3 RW >` 代表该用户对文件 file2 有读的权限，对 file3 有读写的权限。而凡是不在这张表上的文件该用户将不具备访问资格。图 18-3 显示的是 3 个用户和 3 个文件情况下系统维护的 3 张能力表。

在图中，用户 A 的能力表为 `F1 : R; F2 : R`，这意味着用户 A 可以对文件 F1 和 F2 进行读访问。用户 B 的能力表为：`F1 : R; F2 : RW; F3 : RWX`，意味着用户 B 可以对文件 F1 进行读访问，对 F2 读写访问，对 F3 进行读写执行访问。同理，用户 C 对文件 F2 有读的权限，对 F3 有读和执行权限。从图中我们也可以看出，用户 A 不能访问文件 F3，用户 C 不能访问文件 F1。

这样，在每次访问前，操作系统将检查这个能力表来确认一个用户是否有权执行其所要求的操作。如果该用户所要访问的文件在其能力表里，且其权限与所要求的操作匹配，则操作将被允许。否则，该操作将被终止。

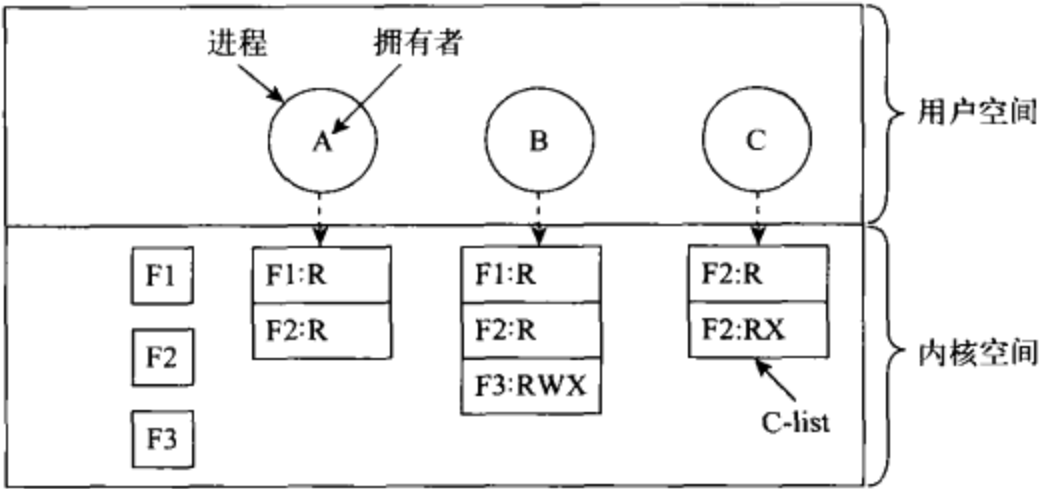


图 18-3 能力表

与访问控制表一样，能力表也是保存在内核空间，由操作系统进行设置与访问。用户不能直接修改能力表。

当然，能力表里的文件名不一定非是个体文件名，也可以是文件集合。例如，图 18-3 里面的 F1 完全可以是一个文件夹名。这样，用户 A 对文件夹 F1 里面的所有文件具有读的权限。这一点对于读者来说应该不奇怪。我们前面已经说过，文件夹就是文件，可以像对待文件一样对待文件夹。因此，对文件夹设置访问控制权限表就没有任何奇怪之处。

而且，这里的用户也不一定非是个体用户 ID，而可以是用户集合，即用户组。例如，图 18-4 显示的就是用户组 faculty 的能力表。

用户	能力表
Faculty	F1:R; F2:rw; F3:RWX

如果把文件看作一辆辆汽车，对文件的访问权限看作是汽车发动机的钥匙，则能力表代表的是一个用户的汽车钥匙集合。如果一个用户拥有特定文件的权限，就相当于一个人拥有特定汽车的发动机钥匙，就能够开动汽车。

能力表的优缺点

能力表的优点是效率高。如果用户要访问的文件是能力表所指向的文件，则无需进行任何检查。其次，能力表有较好的封装。用户和他所能够访问的文件存放在同一个列表里面。而缺点则是删除文件对象困难。如果要禁止所有用户对某个文件的访问，则需要对所有用户的能力表进行遍历。

不过，与访问控制表一样，能力表也可以被黑客攻破。比如说，黑客可以伪造一个能力表。而为了保护能力表，我们可以将其置于操作系统的管辖内。例如，使用 tagged 结构。tagged 结构就是在每个内存字设置一个额外的字位，用来表示该内存字是否包括能力。如果该标志被设置，则对该字的写操作只能由操作系统进行。例如，IBM 的 AS/400 就使用了这种 tagged 结构来实现能力。显然，这种实现需要硬件的支持。

我们也可以对能力表进行加密。这样可以将能力表置于用户空间，但用户需要解密密钥才能使用能力表，从而防止篡改。

18.4 访问控制的实施

有了访问控制表或能力表，或者二者，我们就可以对文件访问进行控制。每次一个用户对文件进行访问时，操作系统将打开用户的能力表和其欲访问文件的访问控制表。在访问控制表和能力表不大的情况下，操作系统通常直接使用这两种手段。但如果访问控制表或能力表非常大，那么执行访问控制的时间成本将增大。这个时候就需要进行某种优化以控制访问控制表或能力表所需的时间。这个手段就是保护域。

保护域

访问控制表和能力表的一个共同缺点就是针对个体的文件需要设置个体的访问控制。如果某些文件或对象的访问控制权限一样，这种个体区分的记录方式就显得有点浪费了。这个时候如果使用保护域就可以解决这个问题。

保护域就是将访问控制权限一样的文件和对象组织成同一个域。而访问控制权限与每个域直接相关。每个域的控制独立于其他域。每个进程都运行在一定的域里，从而具有访问该域里面文件和对象的权限。图 18-5 描述的是具有三个保护域的系统。

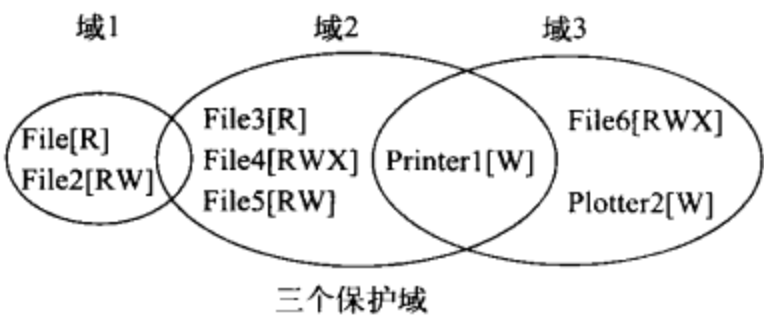


图 18-5 拥有三个保护域的文件系统

在图 18-5 中，file1 和 file2 属于一个保护域，file3、file4、file5 和 printer1 属于一个保护域，file6、printer1 和 plotter2 又属于一个保护域。一个进程在运行时必须处于某个域里。且在运行过程中可以改变保护域，如从用户空间切入到内核空间。

那么系统如何跟踪保护域及其相关权限呢？一个自然的措施是使用矩阵：矩阵的行代表域，列表示文件和对象。例如，图 18-5 的保护域系统可以用图 18-6 的矩阵来表示。

	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
域 1	读	读写						
域 2			读	读写执行	读写		写	
域 3						读写执行	写	写

图 18-6 不包括保护域的保护矩阵

进程运行中的域切换也可以用矩阵实现，例如图 18-7 就表示了运行在域 1 的进程可以切换到域 2(以进入表示)。

使用保护域的优点是可以将保护权限相同的对象组成一个集合来进行处理。缺点是实现方式可能浪费空间。例如，在图 18-6 和图 18-7 中有许多的空白域。如果这种浪费不可忍受，我们总是可以回到访问控制表和能力表两种办法上来。

	File1	File2	File3	File4	Printer1	Plotter2	域 1	域 2
域 1	读	读写					进入	进入
域 2			读	读写执行	写			
域 3					写	写		

图 18-7 包括保护域的保护矩阵

需要特别提醒的是，本章论述的访问控制机制并不是只能应用于文件，也可以用来对其他计算机里的数据对象（即内存对象）进行保护。

18.5 文件系统性能

确保了地址独立和地址保护就实现了文件系统。但是实现了文件系统并不是一切就算结束。除了功能外，一个好的文件系统还需满足用户在性能上的要求。就像计算 $1000 + 1000$ 一样，我们可以在 1000 上加上 1000 次 1，结果虽然正确，但是效率无法让人接受。

文件的性能主要体现在两个方面：可靠性和速度。由于文件系统是用来存放数据的，其可靠性自然非常重要。否则存放的数据将没有人敢用。由于磁盘访问比内存访问慢很多，速度问题比起内存访问时要更加重要。

18.5.1 文件系统可靠性

文件系统的可靠性体现在两个方面：持久性和一致性。持久性，经久耐用，就是存放在文件系统的文件一万年都不变。一致性则指里面存放的数据必须是好的。那么什么样的数据是好的数据呢？就是说存放在一个系统里面的数据之间必须一致，且必须符合客观现实。例如，如果某个人的年龄是 500 岁，则这个数据就不是好的，因为它与客观现实相悖。如果一个人的职位为初级软件工程师，工资却是 50 万元，则这两个数据之间不一致，因此也不是好数据。当然，数据不好不一定是文件系统的错误所导致。但作为文件系统的设计人员，我们必须保障不能因文件系统造成数据不好。

18.5.2 文件系统的持久性

要想文件系统保存的数据持久，单单依靠文件系统本身并无法解决。因为不管文件系统本身做的多好，如果磁盘被人丢进大火里，你上面的数据就不会持久了。当然，造成数据损坏的可能性还有很多。那么实现持久性的手段主要是备份和复制。

数据备份就是制作多余的数据复本，并保存在分开的存储介质上。这样，源数据出现问题时，可以使用备份介质上的数据，从而达到数据持久的目的。

从数据的视角来备份的话有两种：一个是逻辑备份，一个是物理备份。逻辑备份就是备份用户选择好的文件或文件夹。在逻辑备份时，备份的基本单位是文件。而备份过程就是将文件一个个复制到备份介质上，如图 18-8 所示。

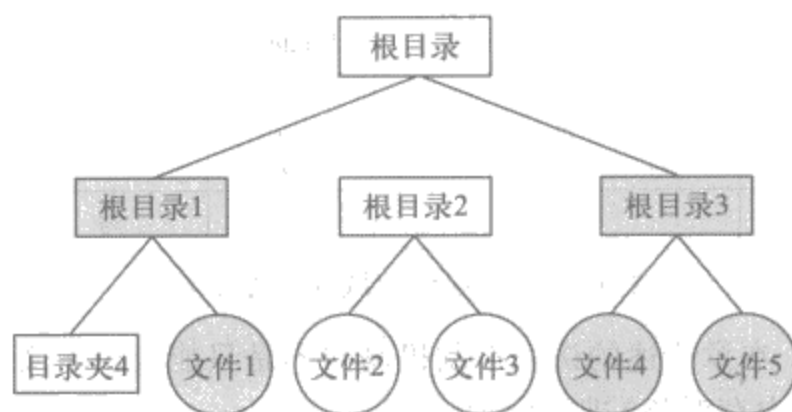


图 18-8 逻辑备份：图中只标有阴影的文件才被备份过去

物理备份则是备份整个磁盘。在物理备份时，备份的基本单位是字节或数据块。而备份过程就是将数据块一个个复制到备份介质上。这里的拷贝是绕过文件系统而进行的，即通过所谓的裸读和裸写而进行数据复制的，如图 18-9 所示。

那么物理备份和逻辑备份哪个更容易呢？多数人都说物理备份更加容易。原因是因为它不需要知道文件系统的结构，只要能够辨认出字节即可。



图 18-9 物理备份：整块磁盘上的数据被裸读到另一块盘上，其结果是两块一模一样的盘

但情况果真如此吗？实则不然。

在现在的市场上，备份产品几乎可以说是不计其数。但绝大部分备份产品都是逻辑备份产品，而不是物理备份产品。这说明什么问题呢？逻辑备份的使用范围广？有可能。但它显示的另一个问题则是逻辑备份比起物理备份来说容易实现一些。

那为什么逻辑备份比物理备份容易呢？或者说物理备份为什么难呢？有几个原因。

首先，是物理备份不经过文件系统，因此需要裸读磁盘，即绕过文件系统对磁盘数据进行读写。而裸读或裸写对很多人来说比在文件系统上读写要困难很多。而逻辑备份使用文件系统进行读写，备份软件无需与磁盘取读者的许多底层细节打交道。

其次，在进行逻辑备份的时候你知道一部分的语意；在物理备份的时候你不知道它的语意，在这种情况下，如果备份中途出现错误你很难侦测到它的错误。即使侦测到，进行恢复也不容易。而逻辑备份则可以文件为单位进行正确性确认。如果出现错误文件只需要重新备份有问题的文件即可。当然，在物理备份时可以数据块为单位进行错误纠正。

最后，物理备份由于是复制磁盘上的一切数据，将造成物理磁盘的卷标也被拷贝。这样将造成物理备份的备份磁盘和源磁盘卷标相同。由于备份磁盘在备份过程中发生卷标改变，将造成备份磁盘无法工作。另外，如果两个卷标相同的磁盘接入到同一部计算机，将造成系统故障或瘫痪。

根据备份方式的不同，备份可以分为很多种类。例如，备份方式有下列几种：

- 主动和被动备份：以谁提要求来区分。
- 在线和离线备份：以是否停止服务区分。
- 实时和延时备份：以主备数据时差区分。

- 等分和差分备份：以主备数据量差区分。
- 增量和差分增量：以备份量和差分标来区分。
- 分裂和并列备份：以备份是否减少来区分。
- 近程与远程备份：以主备中心距离区分。

由于备份是一个很大的课题，并不是文件系统本身的研究范畴，本书就不再赘述。有兴趣的读者可以参阅作者的著作《有备无患：信息系统之灾难应对》一书。

18.5.3 文件系统一致性

前面说过，一致性要求数据是好的。而好的则意味着数据必须彼此之间保持一致，且不与客观现实相悖。这里需要强调的是，数据是好的与数据是最新的是不同的事情。最新的数据不一定是最好的数据，最好的数据不一定是最新的数据。例如，假定某个国家换了总统，但网页上还是前一个国家总统，那么这个数据还是好的，但不是最新的；或者网页上是新的国家总统，这个数据是最新的还是好的。什么样的数据是不好的？就像你说没有国家总统或者有两个国家总统显现在网页上。

美国有一部收视率很高的电视连续剧，叫做《星际探索》(*Star Trek*)。该连续剧里面常常出现一种远程传输(Teleport)设备，能把人从一个地方转到另外一个地方。就是把人分解为分子，发送到另一个地方后将所有分子再重新组装。如果人分解或发送到一半的时候机器死机了，那么组装出来的是什么东西就没有人知道了。不过可以肯定的是，组装出来将不会是正常的人。

如果文件系统设计不佳，其后果就有可能像远程传输设备出现故障一样。不过其后果不是组装出一个人不人鬼不鬼的东西，而是有可能造成数据不一致。

例如，你到银行转账，从你自己账户上转 10000 元钱到左怡的账户上。但这个操作并不是一步可以完成的，它通常分为两个步骤：

- 1) 从自己账户上减去 10000 元。
- 2) 把左怡账户上增加 10000 元。

如果系统在上述两个步骤中间崩溃，则将出现你的账户已经减少了 10000 元，可是左怡却没有得到钱。那么这个时候就出现数据不一致的情况，即数据是不好的。(当然，银行这个时候很高兴，因为银行得到钱了)。

不要以为上述情况不会在现实中出现。2007 年下半年，中国股市火热之时，人人都蜂拥到股市上炒股。但中国有个规定是现金必须保存在银行，只有在需要买卖证券时才将现金转到证券公司账户上，即所谓的银证转账。有一个老年妇女在中国建设银行进行银证转账时就出现了银行账户的钱已经减去 30000 元，而证券账户上却没有收到钱。银证转账一个星期后还是这样。以至于这个老年妇女跑到中国建设银行大吵大闹。

很显然，出现了这种情况，用户将十分的不高兴。当然，如果将上述两步操作反过来，出现中途崩溃时将造成银行损失。这样银行将很不高兴。当然了，在现实生活中，出现问题时总是用户吃亏。因为银行软件总是确保在出现问题时，必须保证银行不吃亏。例如，如果总是先减去客户的钱，再增加客户的钱，就可以保证在任何情况下银行都不吃亏。

再来看我们文件系统里一个非常频繁的操作：建立一个新的文件。

那么建立一个新的文件需要哪些操作呢？有两个操作：

- 1) 你得建立文件的 I-NODE，将 I-NODE 写入磁盘。
- 2) 将 I-NODE 的磁盘地址和文件名写入该文件所属文件夹里。

如果在这些操作中间发生系统崩溃，则文件系统就有可能出现不一致。例如，如果在将 I-NODE 写入磁盘后，但把 I-NODE 的磁盘地址和文件名写入该文件所属文件夹之前发生系统故障，则将出现所谓的“孤儿文件”。即文件不在文件系统里出现（目录里没有），但却在磁盘上。如果将上述两个操作颠倒，则有可能出现“魅影文件”，即文件在文件系统里出现（在目录里有），却没有在磁盘上。

孤儿文件和魅影文件的危害很大。对于魅影文件来说，其问题有三。首先就是文件夹里指向了一个没有被写入的 I-NODE，如果对魅影文件进行读写，读写的东西将是不可预料的，而这种不可预料有可能造成数据的损毁。另外，我们无法确定这个 I-NODE 里面的内容有没有被改写过。还有就是文件夹能容纳的目录项是有限的。那么魅影文件占用了文件目录项，从而使得真正有用的文件可能没有地方存放。对于孤儿文件来说，其问题是无法访问和占用磁盘空间。

又例如，我们移动一个文件的时候有两步操作：

- 1) 将文件从原来的文件夹里删除。
- 2) 将文件加到新的文件夹。

如果在上述两个步骤之间发生问题，则也将出现孤儿文件。如果将上述两个步骤颠倒，则有可能出现两个文件复本，即所谓的“幻影文件”。

不管是幻影、魅影还是孤儿，都表明文件的一致性被破坏。那么如何保证文件系统的一致性呢？答案是使用日志、事务和随影。

1. 日志

日志(logging)就是将文件操作全部记录下来，存放在一个不同的地方，这个地方称为日志。当这些新数据全部写完后，在日志最后写上“完毕”(commit)。在这之后，我们就可以将日志里面的数据往源文件里写。这样就能确保文件数据的一致性。

例如，如果在写入“完毕”前的任何时候出现故障，则新的数据将被丢弃，源文件的数据保持不变。即原来是一致性的数据仍然保持一致。如果在写入“完毕”后发生故障，例如在将数据往源文件里写的中途出现故障，造成只写了一半时怎么办？这也不是问题。在系统恢复后，将新的数据重新往源文件里写就是。可能有人会认为将日志写两遍会造成操作被执行两次而出现问题。但这是不会的。因为日志里面记录的是操作结果，而不是操作。而结果数据写两遍还是一样的结果，因此不会产生任何问题。

2. 事务

另一种确保文件系统一致性的手段是使用“事务”(transaction)。一个事务是一组操作。事务机制可以确保这一组操作要么全部发生，要么一个都不发生。例如，如果有下面一个事务：

```
Begin transaction
- disk write 1
- disk write 2
- disk write 3...
end transaction
```

则磁盘写操作 1、2、3 要么全部发生，要么一个都不发生。事务实际上就是将一组操作转变为原子操作，就像我们讲过的进程同步原语一样。只不过这里的同步是磁盘操作而已。

那么事务是如何实现的呢？答案是显然的，与日志一样。即将所有的磁盘写操作记录在与源数据分开的地方。在完毕后，即可以将这些数据往源数据上复制。

细心的读者可能会发现，事务的开始与结束有点像锁同步机制的获得锁和释放锁。锁的两个操作是 lock 和 unlock，事务的两个操作是 begin 和 end。在 lock 和 unlock 之间的代码可以保证不会被别的线程执行，而在 begin 和 end 之间的磁盘写操作则保证要么全部完成，要么皆抛弃。但是事务是如何确保这种要么全部、要么没有的语义呢？

本书在讨论锁的时候讲过，锁的实现需要硬件支持。事务的实现也不例外，也需要硬件的支持。而这里的硬件支持就是一个扇面的读写。磁盘系统读写一个扇面的操作是一个原子操作，即该扇面要么全部写完或读完、要么就根本没有进行读写。这个硬件原子操作是由磁盘制造商提供的。有了硬件制造商提供的这个操作，事务的原子性就很容易实现了。在事务处理的最后一步是 commit 操作。这个操作就是在磁盘上写入一个特殊的标志，告诉系统该事务处理正常完成。由于这个标志很小，所占空间少于一个扇面，其读写将具有原子性。如果我们看到有这个标志，则事务里面的所有读写操作都已经顺利完成，我们视其为有效。否则，所有操作均视为无效，即以前的操作全部作废。

3. 随影

日志和事务虽然可以提供数据一致性保障，但它们有一个共同的问题：就是磁盘写操作均记录在一个不同的地方，在 commit 后需要将数据复制到源数据处。而在往源数据复制过程中，源数据将处于不一致状态。需要等到数据复制完毕，源数据才归于一致。这样，在复制中，源数据就不能被使用，凡是使用源数据的应用均应该暂停服务。如果复制过程中出了问题，则这种暂停的时间将更为漫长。对于某些应用来说，这种漫长的暂停是完全不能接受的。

而改变此种问题的解决办法就是随影（shadowing）。随影的原理很简单：保持两个数据版本。通过一个指针告诉用户和应用软件当前使用的是哪一个版本的数据。在需要更新数据时，将更新写到非当前使用的版本上。在更新完毕后，修改指针，使其指向新的数据版本。之后在切换下来的数据版本进行更新。这样就能保障数据的一致性。

随影技术主要应用于数据库系统。由于新旧版本切换只牵扯到一个指针的切换，这种切换将十分迅速。由此造成的应用暂停将可以忽略不计。而且，指针切换本身是一个原子操作，不会出现指针破损问题。这是因为指针的大小不会超过一个磁盘扇面的尺寸，而磁盘的一个扇面读写由硬件保证为原子操作。

随影的好处也不是没有代价的，而这个代价就是额外的磁盘空间。因为要保留两个数据版本。

4. 文件系统和用户文件

很显然，确保文件系统的一致性是要付出代价的。日志、事务和随影都需要使用额外的空间和时间。如果每次文件操作均需要使用这些方式来保护，则文件系统的效率将显著下降。但我们又不能不保护文件系统的一致性，因为这是文件系统正确性的前提。

那有没有什么办法既可以确保文件系统一致性、又能够不显著降低系统效率呢？

有。那就是我们只是有选择地对某些文件进行一致性保护，而对另外一些则不予保护。那么我们要选择的文件必须是对文件系统非常重要的文件。那么什么文件对文件系统很重要啊？当然是文件系统的元数据（metadata），也就是文件夹、自由空间表格等。

因为这些元数据的丢失将造成文件系统不能正常运行。因此，我们对其进行保护。而对于用户数据则不进行一致性保护。因为用户数据损坏影响的是单个的用户或文件，并不会影响整个文件系统。而文件系统本身出现故障将影响所有的用户和文件。如果用户很关心自己数据的一致性和安全性，用户应该采取数据备份或容灾手段予以确保，而不应该依赖文件系统。

当然了，如果在效率和复杂性允许的情况下，照顾一下用户的数据也不是不可以。但作为一个文件系统的设计者，我们的首要目的是确保自己设计的东西都正常运转。因为用户数据丢了可以让用户再输入一遍。而如果源数据丢了，则很有可能无法恢复。这就像打战一样可以牺牲少数人，只要多数人能得到保护就可以。

这样，我们就可以一方面确保文件系统的一致性，一方面又不显著降低效率。

5. 文件系统一致性检查

前面几节讲解了文件系统一致性以及如何保证文件系统一致性。那么如何判断一个文件是否一致，是否完整呢？

要进行这种判断，首先需要知道文件系统一致性到底体现在什么方面。或者说什么状态是一致状态？

对于文件系统来说，只要文件系统能够正常运转就说明文件系统处于一致性。当然，这不包括用户数据的一致性。具体来说，文件系统一致性包括如下几个方面：

- 没有魅影文件。
- 没有孤儿文件。
- 文件 I-NODE 里面的链接计数与其出现在文件系统中不同位置的次数一致。
- 没有消失的空间。
- 没有额外的空间。

因此，对文件系统一致性进行检查就是对上述 5 个方面进行确认。而要对这些进行确认，当然需要保持足够的信息。对于前三点来说，这些信息保存在文件目录和 I-NODE 里面。

后面两点的意思就是一个磁盘块要么在闲置空间里，要么被占用，而不能同时处于两种状态。为了对后面两点进行判定，我们需要维护两张表：一张闲置空间表，一张占用空间表。通过两种表进行相互比对来判断是否存在消失或额外的空间。例如，图 18-10 所示的两种表表示一切正常。一张表为 0 的位对应另一张表为 1 的位。说明该磁盘块只存在于一张表里面，即要么它处于闲置状态，要么它已经被分配给某个文件使用，而不是同时处于两种状态。

闲置空间表															
1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1
占用空间表															
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

图 18-10 系统一致状态下的闲置空间表和占用空间表

而图 18-11 所示的情况则说明出现了问题，有一个磁盘块对应的两种表的位置皆为 1，说明，该数据块既是闲置，又被占用，即出现了所谓的“一仆二主”问题，这在正常情况下是不可能的。出现这种现象说明文件系统出了问题。

闲置空间表															
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1
占用空间表															
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

图 18-11 “一仆二主”状态下的闲置空间表和占用空间表

什么样的情况下会发生这种状况呢？如果系统释放一个磁盘块，我们首先将闲置空间的对应字位设置为 1，表示该空间已经释放。然后我们需要将占用空间表里的对应字位设置为 0。但在设置为 0 前，系统因故不能继续，从而造成图 18-11 的问题。

图 18-12 所示的情况也说明出现了问题，有一个磁盘块对应的两种表的位置皆为 0，说明，该数据块既非闲置，又未占用，即这个数据块消失了。这就是所谓的消失的空间问题。而出现此种现象的原因也可以类似推出。

闲置空间表															
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1
占用空间表															
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

图 18-12 消失的空间状态下的闲置空间表和占用空间表

如果我们只维持一张表，即将空间闲置表和空间占用表合二为一，1 表示占用，0 表示闲置，则在系统发生错误时将无法判断。例如，如果一个闲置磁盘块被误置为 1，则这个磁盘块将不会再被使用，从而造成消失的空间问题。而这个问题因为只有一张表而无法得到诊断。如果一个占用磁盘块被误标为 0，则将造成数据块被覆盖的危险，同理，因为只有一张表，系统将无法对此种问题进行判断。

18.6 提高系统性能的方法

不管做什么，性能总是我们要考虑的一个因素。我们在第 11 章里面就论及到性能。

TLB 和缓存就是为了提高内存管理的性能或效率而设计出的机制。那么在文件系统里，性能自然也是非常重要的，甚至于比内存对性能的重视程度更高，因为文件磁盘访问比内存访问慢多了。

当然，最能够想到的提高磁盘访问速度的办法就是提高磁盘自身的访问速度。即制造更高速度的磁盘。但这种方法不是我们研究操作系统的人所能控制的，而是由磁盘制造商决定。而且，磁盘的旋转速度确实在稳步的上升，从一开始的 1500 转到 3600 转到 7200 转再到 20000 转，应该说已经提高了很多。旋转速度的提高既降低了旋转延迟，又提高了数据传输效率。但是磁盘访问的瓶颈并不是旋转延迟，而是寻道延迟。而寻道延迟与磁盘旋转速度没有关系。影响寻道延迟的因素是磁盘的机械磁臂的移动速度。虽然磁臂做的是直线移动，而直线运动可以做的很快，但对于磁臂来说却没有什么意义。这是因为磁臂寻道时需要准确地停在需要的磁道上。如果磁臂运动太快，将不能精准的停留在所需磁道，从而需要来回调整，反而变得更慢。因此，磁臂的移动速度很难不断地提升，即磁盘自身访问速度的提升是有限的。

既然磁盘本身的速度提升有限，又不属于操作系统设计人员能够控制，那还有什么别的方法可以提升磁盘访问速度呢？我们在进程管理和内存管理部分讨论过几种提高系统性能的方法：合适的调度、缓存、提前读取，而这些方法可以应用到文件系统上来：

- 将经常访问的内容置于缓存里。
- 减少磁盘访问时磁臂移动的距离。
- 提前将需要的数据块读入内存。

18.6.1 文件缓存

文件缓存就是将文件内容的部分置于缓存，从而可以从缓存满足用户的文件访问请求，而无需每次都到磁盘上去读写。这样可以大大提高访问效率。而放入缓存的数据既可以是用户数据，也可以是元数据。由于元数据的使用频率很高，例如，文件系统的自由空间表格、文件头、间接数据块、文件夹等数据结构在整个计算机系统运转过程中需要经常使用，缓冲元数据比缓存用户数据效果将更加明显。

那么文件缓存的实现有两种方式：

- 1) 缓存在虚地址空间。
- 2) 缓存在实地址空间。

缓存在虚地址空间就是缓存在虚拟内存里，即将文件映射到进程的虚拟地址空间，即我们前面讲过的内存映射的文件。这样的话对文件的访问将变为对内存的访问。这种方式的优点就是简单，因为我们将工作交给了内存管理单元，即 MMU。但缺点则是不一定能够提高文件访问效率。因为 MMU 不一定会把我们需要的文件保存在内存里，它随时可能被交换或替换出去。另外，如果有多个进程使用该文件的话也会出现问题。

缓存在实地址空间就是缓存在物理内存里。此种方式需要由文件系统自己负责管理。即文件系统需要决定什么数据需要缓存，什么数据需要更换等。这样自然导致文件系统将变得复杂，但是可以确保需要的数据总是存放在缓存里。

18.6.2 虚拟内存和文件缓存

那么使用文件缓冲后，文件的部分将处于内存、部分处于磁盘上。这就和虚拟内存（VM）很相似：一部分程序处于内存、一部分处于磁盘上。因此从磁盘的角度来看，虚拟内存和文件缓存（FC）看起来似乎一样。那我们何不将二者结合起来呢？例如，我们可以将文件缓存交给 VM 来管理，或者将 VM 交给文件缓冲来管理。

要回答上述问题，需要知道 VM 和文件缓存的根本区别是什么。VM 的根本目的是提供一个速度非常高、容量非常大的并不存在的内存空间。它从物理内存出发，为了增加内存空间而扩展到磁盘上；而文件缓存是为了提高文件的访问效率而出现。它从磁盘出发，为了提高访问效率而将文件置于缓存。就是说 VM 和文件缓存的根本目的不同，即有着本质的区别。因此，其工作机制、语义和操作界面皆不相同。

而正是由于这些机制、语义和操作界面的不同，使得它们不能相互替代对方的角色。例如，文件访问请求发出的是磁盘地址，如磁盘数据块编号。VM 能辨别出磁盘地址吗？它知道一个磁盘块是多大？从什么地方开始？从该磁盘块开始如何找到下一个磁盘块？显然不能。而对于内存访问来说，其发出的是虚拟内存地址，试问文件系统认识这种地址吗？它知道页面号占几位？段位占几位？它怎么知道如何将虚拟地址转换为物理地址？当然不能。

也许读者会说，那可以把 VM 设计得能够辨认文件访问地址，或者将文件缓存设计得能够辨认分页、分段系统。理论上讲这样是可以做到的。但实际操作起来非常困难。首先，因为这样做将要求 VM 能够辨认所有的文件系统，因为你不清楚在实际环境中到底哪种文件系统会被挂载，而这是几乎不可能的。因为在 VM 设计实现后，人们还在继续开发新的文件系统。你不能不停地修改 VM，更新操作系统来适应这些变化。而要把文件系统修改成分页或分段系统则更为困难。本来这个文件系统的可靠性我们就不满意，现在你再加了这么个功能，又大大降低了文件系统的可靠性。

其次，这些修改将使系统变得非常复杂，而复杂的东西就不容易做的可靠。而最重要的障碍就是违反了层次分离的原则。

18.6.3 提前读取

提前读取就是在每次访问磁盘时，我们多读一些数据出来。比如用户要求读取 2 个数据块，我们实际上读出 10 个数据块。因为按照时空局域性的原则，用户访问一个地方后，很可能会访问该地方后面的地方。因此我们提前将其后面的数据读出来。这样在用户随后需要访问这些数据的时候就可以从内存得到满足，从而提高文件访问效率。

由于磁盘访问的瓶颈是寻道和旋转延迟，而数据传输相对来说所占消耗量很小。这样读出一个数据块和读出 10 个数据块所费时间几乎没有任何区别。因此，提前读取的代价几乎为 0。

当然，提前读取只在用户对数据是顺序访问或者准顺序访问时效果才好；对于完全随机的访问效果则不太显著。

18.6.4 减少磁臂移动

磁盘访问的瓶颈是其机械运动部分。即磁臂移动和磁盘旋转。而磁臂移动又占主导地位。我们前面说过，受运动精度限制，提高磁臂移动速度的空间有限。那么剩下的节省时间的办法就是缩短磁臂移动距离。而缩短磁臂移动距离有两种办法：

- 将文件头（I-NODE）和数据块放置在一起。
- 将文件读写操作进行适当排序。

在讲文件布局时我们说过，I-NODE 是单独放在一起，与文件数据是分开的。这样就造成读 I-NODE 和读文件数据之间需要进行磁盘寻道。而第一种办法的考虑因素是文件打开时需要读取文件头。而一般来说，文件头读取后立即或很快就需要访问文件内容。如果 I-NODE 和数据块放在一起，读取数据块时就不需要寻道。这样就会大大提高读取数据的速度（见图 18-13）。

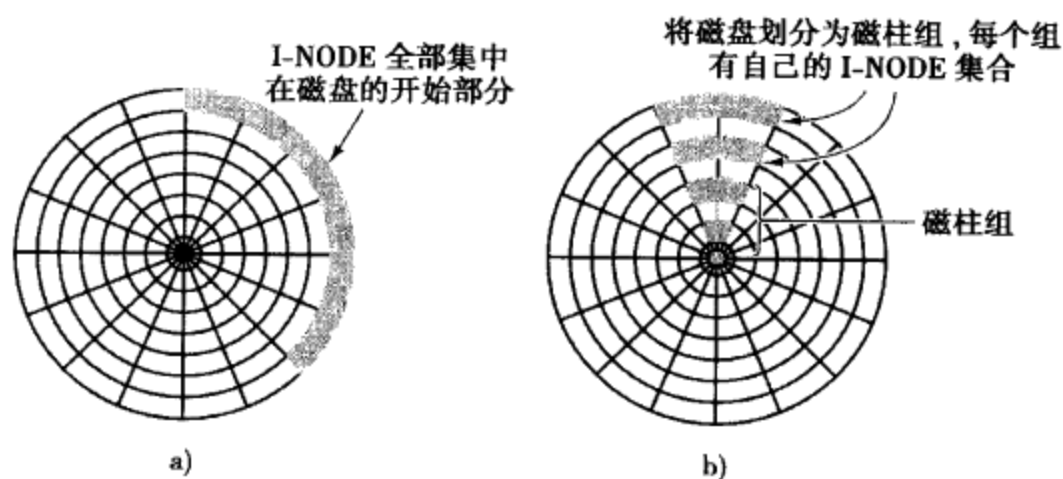


图 18-13 I-NODE 集中存放与分散存放（来源参考文献 [3]）

不过，将 I-NODE 和文件数据块放在一起会对文件一致性检查和清空磁盘的操作造成麻烦。由于在进行磁盘数据一致性文件检查的时候需要访问所有的 I-NODE，如果它们放在一起就效率高，而分开放置需要一个一个 I-NODE 的找，效率就会很低。在删除磁盘上所有文件的时候也需要访问所有 I-NODE。显然 I-NODE 的分散放置将降低此种操作的效率。

图 18-13 描绘的是 I-NODE 集中存放和按组分开存放的两种模式。

第二种方法是将一系列的磁盘访问操作按照磁臂最小移动的原则进行排序。使得从一个访问到下一个访问的磁臂移动距离最短，从而提高磁盘的整体访问效率。关于磁盘访问的调度已经在本书第 15 章论述过，这里就不再赘述。

18.7 文件系统设计分析：日志结构的文件系统

下面我们讨论一个文件系统设计案例，以加深对本章讲解内容的理解。这个文件系统就是所谓的日志结构的文件系统（LFS，Log-Structured File System）。

日志结构的文件系统是美国 UC-Berkeley 提出的一种文件系统。其目标是提高文件系统的效率。其工作原理的出现依赖于以下的观测：随着 CPU 速度的提高和物理内存空间的增長，

磁盘缓冲可以做的更大,使得越来越多的读操作可以从缓存得到满足。因此,大部分磁盘访问实际上只是磁盘写操作。

因此,要提高磁盘访问效率,就需要提高磁盘的写操作效率。那么磁盘的写操作效率的瓶颈在什么地方呢?当然是寻找适当的写位置。那么如何改善此处的效率呢?显然,如果我们在磁盘写操作时不用寻找写的位置,则磁盘写操作效率将大大提高。

UCB正是看到了这点而设计出了日志结构的文件系统。在该系统下,整个磁盘被当作一个巨大的日志看待。所有的磁盘写操作从日志的尾端开始。这样就避免了寻道操作。因为我们刚才说过,磁盘访问主要是写操作,而每次写从上次写毕的位置直接开始。当写到磁盘末尾时,我们再从磁盘头接着写下去。即将磁盘看作是一个循环日志。

而为了进一步提高写的效率,我们还结合缓存来进行。即不是将所有的写操作随时进行。而是将写操作在内存里缓存起来。然后在系统闲暇时,或者周期性地,将缓存起来的写操作写入到磁盘日志上。由于每次都写在新的地方,我们需要将I-NODE和文件内容一起写入。

由于I-NODE所处的位置随着日志的写入发生变化,LFS使用一个I-NODE表来记录每个I-NODE的当前位置。每次需要访问一个文件时,从目录里找到该文件的I-NODE位置,然后读取数据块并将其置于文件缓存里面。

由于每次文件修改后的内容都写入到日志的末尾,即为该文件创建了一个新的版本,将造成一个文件在磁盘上存在多个版本。这样,磁盘空间将很快就会占满。下次需要空间时就会没有空间。为解决这个问题,LFS使用一个垃圾清理程序来清理已经被废弃的文件。当然,这个垃圾清理程序只在磁盘闲暇时或者磁盘空间已满时才运行。

那么LFS有没有提高文件系统的效率呢?要回答这个问题并不容易。

从表面上看,文件系统效率是提高了。因为读从缓存满足,写的时候又不需要寻道,从而使得读写效率都获得提高。

那么LFS文件系统有没有什么缺点呢?当然有。天下没有免费的午餐。根据前面所述,LFS效率的提高依赖于多个条件的满足:

- 绝大部分读操作可以从内存满足。
- 连续的写之间不会插入磁盘读操作。
- 磁盘的自由空间为连续的。

我们刚才说过,LFS效率提高的一个条件是磁盘的绝大部分读操作能在内存中满足。当时UCB做了很多试验,当读的80%在内存满足时,LFS的效率提高就会较为显著。但是只要你的读不能在内存中满足,那么这个文件系统的优点就不能体现。那么读操作当真可以绝大部分在内存满足吗?这一点似乎不难做到。因为,每次打开文件时可以一次性将文件全部读到内存。这样以后对此文件的读操作皆可以从内存满足。

但是,文件缓冲真的像UCB声称的那样可以满足绝大部分读请求吗?这实际上并不一定。这得看一个系统同时访问的文件夹数量有多少?文件的大小有多少?读文件的目的是什么?这些因素可能使得UCB的声称大打折扣。

第二和第三两个条件是保证每次写操作时无需寻道。如果在两次写日志之间发生磁盘读操作,则磁盘很有可能移动了位置,从而造成下次写操作时需要进行寻道。如果磁盘空间不是连

续,则需要跳跃写,也将发生寻道操作,从而导致写操作效率的下降。那么这两个条件是否可以保持满足呢?这可不一定。谁能保证磁盘的写操作之间不发生读磁盘操作?很有可能我在写过一个文件后,需要打开另一个文件进行读操作。这样将造成磁头偏离日志的尾部。而随着每次文件写入,都会产生一个新的版本,从而使前面的版本作废。问题是磁盘上可能作废的文件很多,但是磁头下面要写的部分却是没有作废的文件,这将严重妨碍 LFS 的顺利运转。

而且,由于 I-NODE 和数据块的位置在每次写操作时皆发生变化,I-NODE 和文件夹的更新将比一般的文件系统更为频繁,管理起来自然更为复杂。另外,由于 I-NODE 分散存放,文件系统一致性检查和删除磁盘内容的操作将变得费时。

18.8 海量数据文件系统

随着计算机应用的不断扩大,文件系统所处理的数据量呈爆炸增长态势。当文件系统里面数据量增长到一定的程度,传统的文件系统将无法有效运转。例如,在 NTFS 下,如果单一目录下的文件数量超过十万个,文件系统的性能将急剧下降,达到几乎无法运转的地步。因此,针对大数据量的情况,文件系统的构造必须发生变化,即必须针对海量数据专门构造所谓的海量文件系统。

从原理上看,海量文件系统与普通文件系统并无太大区别,它所应该达到的目标仍然是磁盘抽象,提供地址独立与保护,但在性能上的挑战非常明显。挑战是因为数据量的增长突破了某一阈值,而这一阈值的突破导致传统文件系统的失效。这就是哲学上量变质变的规律在文件系统上的体现。

那么海量文件系统的构造与传统文件系统的不同在什么地方呢?要理解这一点,先要看看海量文件系统的几个特点:

- 数据量大:由于数据量巨大,在巨大的目录里面迅速找到所要的文件(文件头及其数据块)不是一件琐碎的事。
- 分布环境广:海量文件系统通常构建在分布式环境下,由于用来提供文件系统存储介质的可能是多个磁盘,且可能位于不同的节点上,如何判断所需文件位于哪个节点并迅速获取数据就变得很重要。这与在单一磁盘(阵列)上构建的普通文件系统相差甚远。

基于上述两个因素,海量文件系统在性能上面临巨大挑战。如果仍然使用普通文件系统那种将元数据与用户数据一起放置的办法,则搜索起来将费时费力。而避免这种窘况的办法就是将元数据与平面数据分开存放,从而减少搜索的数据量,提升寻找所需文件的效率。如果元数据量本身也很大,则还需要将元数据分拆到多个节点存放。这种存放元数据的节点就称为元数据服务器。而存放平面数据的节点就是所谓的数据服务器。

当然,平面数据本身也可以按照某种规律进行划分(不是分区),将不同划分存放在不同的节点上,通过各种算法如散列来定位数据所在的节点。

如果上述手段还没有达到效率目标,或者我们还需要进一步提升处理的效率,则可以将目录数据从元数据中分离,构建专门的目录服务器,从而形成目录服务器、元数据服务器、平面数据服务器的三层架构。例如,Sun 公司的 Lustre 分布式文件系统与 EMC 公司的多通道文件系

统 MPFS 都是采取了元数据与平面数据分开放置的海量数据文件系统。

海量数据文件系统由于数据量变化所带来的挑战当然还有许多，如何构件合理的海量文件系统上未获得统一的认识，分析讨论各种实现方式超出了本书的目标范围，建议有兴趣的读者参阅有关公司的产品介绍和相关论文。

思考题

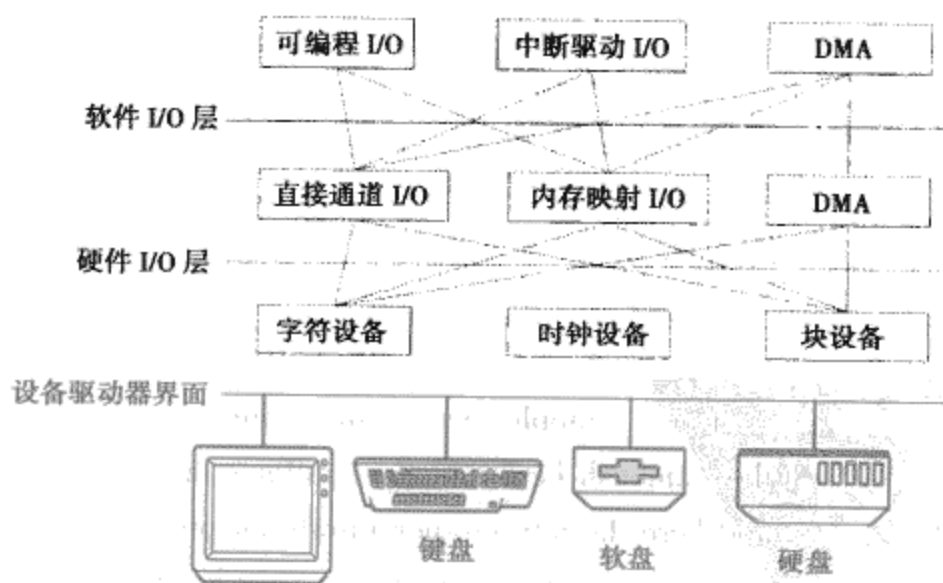
1. 在 ACL 系统里，一个对象的拥有者如何吊销另一个用户对该用户的访问权限？
2. 在能力表系统里，一个对象的拥有者如何吊销另一个用户对该用户的访问权限？
3. 保护域是怎么回事？它与访问控制表和能力表之间存在何种关系？
4. 文件系统一致性是什么意思？如何检查一个文件系统的一致性呢？
5. 文件系统一致性为什么不包括用户文件内容的一致性？
6. 什么情况下文件系统会出现消失的空间问题？
7. 内存映射的文件是否可以作为提高文件读取效率的办法？为什么？
8. 对磁盘进行分区有哪些好处？有什么缺点吗？
9. 设有多个不同的进程共享一个文件。每个进程有一个独特的优先级数。该文件可以同时被多个进程访问，但需要满足如下条件：所有同时访问该文件的进程的优先级数相加必须小于 n 。
 - a) 请问，如果使用管程实现同步，条件变量是什么？
 - b) 请同时给出伪代码的实现（记住在需要的时候使用 wait 和 signal/broadcast）
10. 请仔细比较事务和锁两种机制的异同点。
11. LFS 是通过何种机制提高文件系统的访问效率的？
12. 讨论日志、事务和随影三种维持文件系统一致性机制的优缺点。
13. 讨论虚拟内存和文件缓存之间的异同点，为什么不能把这两种机制合并？
14. 维持文件系统持久性的手段有哪些？
15. 文件缓存和内存映射的文件是否是一回事？为什么？
16. 本章引子里面撒旦的话语存在什么逻辑漏洞？

第五篇 I/O 原理篇

有了计算，又有了存放数据的临时和永久地方，似乎计算机需要的一切都已经具备。但这是真的吗？我们知道，发明计算机的目的是让计算机为人类服务，而计算机要为人服务就得有一种办法与人类进行沟通或者交互。而实现这种交互的手段就是计算机的输入和输出。一台没有输入和输出的计算机，其 CPU、内存和磁盘再强大，对人们也并无太大用处，甚至说是毫无用处。就像一个人一样，如果其能力再强，大脑再聪明，只要他与外界的沟通渠道（听、说、读、写）被打断或有缺陷，就会被世人看做一个无用的人。

因此，要想让计算机真的有用，就得有输入和输出。而操作系统既然是计算机的掌控者，当然也需要对输入输出进行掌控。本篇即对计算机与外界进行沟通的输入与输出机制进行讲解。本篇仅有一章（第 19 章），讨论的内容包括输入输出的重要性和目的、输入输出硬件、物理 I/O 模式（专有通道 I/O、内存映射的 I/O、复合 I/O、DMA）、输入输出软件、软件 I/O 模式（可编程 I/O、中断驱动的 I/O、DMA）、I/O 软件分层、设备驱动程序等。

本篇最为重要的核心是输入输出的硬件模式与软件模式，及这两个模式之间的对应关系。理解了这两个概念，其他部分就自然明白了。



输入输出从根本上说是操作系统对各种外部设备进行的抽象和装扮

第 19 章 输入输出

引子

1709 年，在英国的斯特福特郡 (Staffordshire)，一个小孩出生了。由于贫穷多病，在年幼时即左眼失明、右眼高度近视，并且左耳失聪和脸部畸形。

虽然这个小孩从小就异常坚强，但仍然摆脱不了命运的无情摧残。1728 年，其母亲竭尽全力供他进入了剑桥大学，但一年后，由于贫穷，这个已经 20 岁的年轻人被迫辍学。望着自己唯一一双前露脚趾后露跟的鞋，年轻人陷入了长久的抑郁。他站在镇里的钟楼前面望着时钟却说不出时间，他对外界没有了任何反应。而外界也已经认定这个人是一个十足的痴呆。

就在这个年轻人准备自杀的时候，一个女人却从他痴呆的表象中看到了不同，并毅然地与这个小伙子结合。

奇迹诞生了，小伙子的天才迸发了：

这里的怨恨阴谋和劫掠
变为暴民的狂热与烈火
十足的恶棍做好了埋伏
凶残的律师寻找着猎物
失修的房子在头顶坍塌
女无神论者在口吐白沫……
可哀的真理在各处承认
缓慢起价值，贫穷生堕落。

Here malice, rapine, accident conspire,
And now a rabble rages, now a fire;
their ambush here relentless ruffians lay,
and here the fell attorney prowls for prey;
here falling houses thunder on your head,

and here a female atheist talks you dead……
This mournful truth is everywhere confessed,
Slow rises worth, by poverty depressed.

他在写作上的奇异天赋很快征服了世人，而英国18世纪的后半部分也由他的名字命名。

这个人就是英国杰出的文学家，成就仅次于莎士比亚的赛缪尔·约翰逊（Samuel Johnson）（见图19-1），他的妻子就是波特夫人。英国18世纪的后半部分称为“约翰逊时代”。

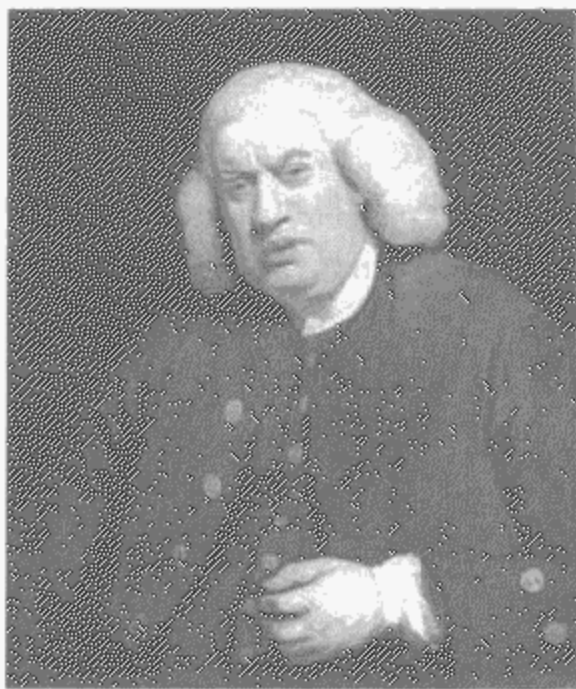


图19-1 英国文豪赛缪尔·约翰逊（1709 - 1784）

19.1 什么是输入输出

本书的前三个部分详细阐述了操作系统的三大核心功能：进程管理、内存管理、外存管理（文件）。但是单有这三个核心功能是不够的。计算机归根结底是为人类服务的，这就要求计算机必须提供某种机制使得人可以向计算机发出命令或操纵计算机。也就是说计算机与人之间必须存在某种沟通的机制。这种沟通的机制就是计算机的输入输出机制。

输入提供的是一个“人→计算机”的通道。即人或外部世界通过输入向计算机发出命令或提供数据。输出提供的则是相反方向的通道，即“计算机→人”的通道。计算机通过这个通道向人或外部世界输出自己的计算结果，包括对其他设备的控制操纵命令。

显然，输入输出的存在才使得计算机的存在有了意义。就像一个人，如果没有输入输出，即他不能与外部世界打交道，则这个人通常被认为是痴呆或白痴。即使这个人实际上是一个天才，情况也是如此。同样，一台没有输入输出的计算机，不管其运算功能多么强大，也是废铁一堆（对于计算机外的世界，或者人类来说）。由此，输入输出也就成为操纵系统设计时的一

个重要考虑。

对于操作系统设计人员来说，从高层设计来看，关于输入输出我们要问的问题有两个：

- 输入输出要达到什么目的？
- 操作系统是如何实现输入输出功能的？

19.2 输入输出的目的

输入输出的目的，从简单来说，就是提供一个人机交互的通道，使得人与计算机能够进行沟通。但这是抽象的层次。具体来说，输入输出的目的是什么呢？

我们在前面讲述进程、内存和文件时说过，操作系统是一个魔术师和管理者。对于输入输出这部分功能来说，也不例外。操纵系统要管理的自然是输入输出设备。而魔幻则是提供一个统一的界面来屏蔽输入输出设备的差异，使得数据的表示能够在不同设备之间相互转换而无需用户的操心。那么到这里，我们就可以得出操作系统输入输出的目的是：

- 屏蔽输入输出设备的差异。
- 在不同设计之间进行数据表示的转换。

达到上述目的需要的机制，仿照我们前面的模式，是：设备独立与设备保护。

这里的设备独立指的是输入输出不以设备的不同为转移，即不管输入输出设备是否更好或更新，我们进行输入输出的模式和方法保持不变。而设备保护则是一个输入输出设备的操作不会影响另一个输入输出设备的操作。

那么要想实现设备独立和设备保护，我们需要从硬件和软件两个方面出发进行考虑。下面我们先来探讨一下硬件层面的输入输出，然后再探讨软件层面的输入输出。

19.3 输入输出硬件

输入输出设备种类繁多，功能各不相同，操控也不尽相同。对于普通人或者电气工程师来说，输入输出设备呈现的首要特征是其物理组件：芯片、布线、能量供应、电机等诸如此类的东西。而对于软件工程师或程序员来说，输入输出设备呈现的则是程序员或用户界面：可接受的命令、能提供的功能、错误处理机制等。毫无悬念，本书采取的自然是从软件工程师的角度，尤其是操作系统设计人员的视角来对待输入输出。

从程序员或操作系统设计人员的视角来看，所有的输入输出设备可以（大概）划分为两个大类：块设备和字符设备。这种划分是以设备存储和传输数据的方式来决定。块设备，顾名思义，就是以数据块为单位存储和传输数据的输入输出设备，如磁盘、光盘、U 盘、磁带等。而字符设备自然是将数据按照字符（字节）为单位来存放和传输数据的设备，如鼠标、键盘、打印机、网络界面等。

当然，上述分类并不是绝对的。例如，一个设备可以同时作为块和字符设备。例如，网络界面通常被认为是一个字符输入输出设备，但在某些时候可以与内存进行 DMA，从而看上去

更像一个块设备。而另外的设备，如时钟，则不属于这两种中的任何一种。（时钟是输入输出设备吗？）

字符设备和块设备的最大不同是在寻址。块设备的数据按数据块为单位进行处理，而每块数据块都有一个唯一的磁盘地址。也就是说数据块是可寻址的。而字符设备里的字符是不可寻址的。当然，由于一个字符占一个字节（对于 ASCII 码来说），而字节是可以寻址的，很多人会认为字符因此也是可寻址的。但这个理解是不正确的。（为什么？）

19.3.1 输入输出设备的差异性

输入输出设备由于种类不同，制造商不同，技术标准不同，其特性可以有巨大的不同。而这种不同越是明显，对操作系统的设计的挑战就越大。因为屏蔽这些巨大的不同，使得不同的设备相互共存并转不是一件容易的事情。

其中最为明显的一种差异是其数据传输的速度。输入输出设备的传输速度涵盖范围从每秒十个字节到几兆个字节。表 19-1 给出的是较为常见的一些设备的数据传输速率。

表 19-1 数据传输速率

输入输出设备	每秒数据传输速率	输入输出设备	每秒数据传输速率
键盘	10B	快速以太网	12.5MB
鼠标	100B	ISA 总线和 EIDE 磁盘	16.7MB
56K 调制解调器	7KB	火线	50MB
双向 ISDN 线路	16KB	XGA 监视器	60MB
激光打印机	100KB	SONET OC - 12	78MB
扫描仪	400KB	SCSI Ultra 2 磁盘	80MB
以太网	1.25MB	千兆级以太网	125MB
USB	1.5MB	Ultrium 磁带	320MB
IDE 磁盘	5MB	PCI 总线	528MB
40X CD - ROM	6MB	Sun 千兆平面 XB backplane	20GB

19.3.2 设备控制器

输入输出设备本身并不是一个不可分割的整体，而是由不同的部件构成。一般来说，一个输入输出设备至少可以分为两个部分：机械部分和电子部分。

机械部分自然是设备的物理硬件部分，而电子部分则是设备的控制器。控制器有时也称为适配器，通常为一片印刷电路板。控制器可以处理多个设备，或者说多个同类的设备可以共用一个控制器。图 19-2 描述的就是输入输出设备和它们的控制器。

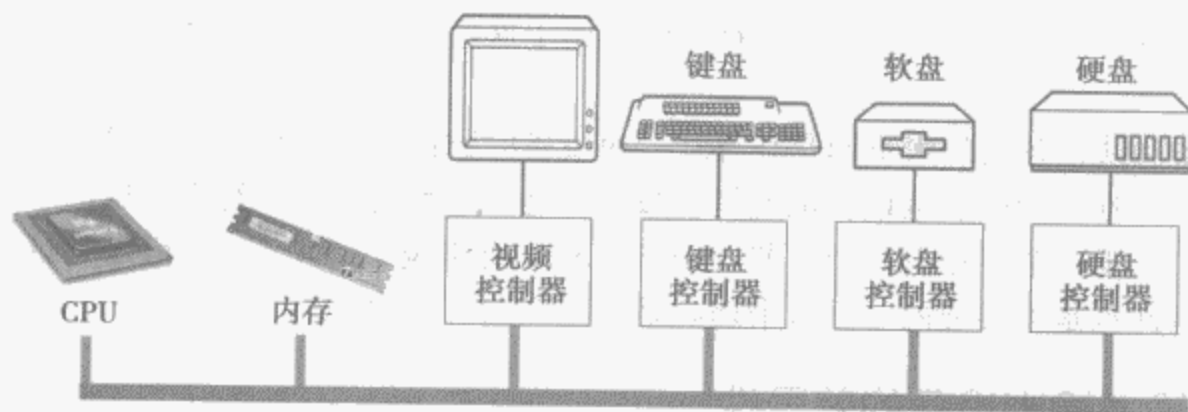


图 19-2 输入输出设备和它们的控制器

设备控制器的任务可以简单地分为如下几项：

- 控制设备的物理运行。
- 将序列字位流转化为字节块流。
- 进行纠错操作。

设备控制器与 CPU 之间的数据交互通过设备寄存器进行。设备寄存器附着在设备控制器上。通过向这些寄存器进行写入，操作系统可以向设备发出输入输出命令，或把设备关闭或打开。而通过读取这些寄存器的内容，操作系统可以获得设备的状态信息。

为了提高与 CPU 交互数据的效率，输入输出设备通常还备有数据缓冲区。例如，视频控制器通常带有自己的视频 RAM。CPU 通过与视频 RAM 进行数据交互，就可以传输巨大的数据量。

19.3.3 物理 I/O 模式

计算机在输入输出时可以选择的模式很多。这些选择的不同体现在 CPU 和内存的使用不同上。通常来说，物理输入模式可以按照两种考虑进行分类：

- 根据 CPU 访问 I/O 设备的方式进程分类。
- 根据 CPU 在 I/O 过程中的涉入程度进行分类。

根据 CPU 与设备控制器沟通方式以及与内存的不同关系，物理 I/O 模式可以分为以下三种：

- 专有通道的 I/O
- 内存映射的 I/O
- 混合 I/O

1. 专有通道 I/O

在专有通道 I/O 模式下，I/O 与内存是完全脱离的。每个控制寄存器被赋予一个 I/O 端口。这个 I/O 端口就是一个 9 位或者 16 位的一个整数。这个整数与内存地址没有任何关系。而正是由于 I/O 端口地址与内存地址没有任何关系，或者说 I/O 端口地址不是内存地址，操作系统必须使用专门的输入输出特殊指令来进行数据的读写。例如，许多指令集结构均使用 IN REG, PORT 指令用来从设备读取数据；而指令 OUT PORT, REG 则用来将数据写入设备。

专有通道模式的优点是与内存分开，输入输出操作不会影响或干扰内存操作。尤其是输入

输出软件的可靠性通常不如内存管理软件，这种分离就显得更加有价值。但俗话说，成也萧何，败也萧何。这个优点恰恰也是其缺点。由于与内存分开，输入输出指令与内存访问指令自然也不相同，因此，进行内存访问与进行输入输出的指令互不相同。事实上，正如上一段所描述的，这种模式下的输入输出需要使用专门的 IN/OUT 指令来进行。

由于高级语言不支持这种低级指令，从而形成高级语言屏蔽的一个漏洞。程序员如果想进行 I/O，则必须使用高级语言，如使用 IN 或 OUT 指令，而这将增大程序设计的难度和可靠性。

专有通道 I/O 还有一个问题，就是保护问题。这是因为由于使用低级语言进行 I/O，而低级语言的保护作用非常有限。例如，低级语言一般不具备访问控制和授权能力，使用这种模式进行访问的主体可以对 I/O 设备进行任意操作，包括刻意制造故障。

早期的计算机使用专有通道 I/O 模式，这些计算机包括 IBM360、370。

2. 内存映射的 I/O

内存映射的 I/O，顾名思义，就是将 I/O 映射到内存里面，从而使得 I/O 和内存管理得到统一。具体来说，就是将 I/O 设备的每个控制寄存器和设备缓冲区寄存器赋予一个唯一的内存地址。对这些地址的访问就是对输入输出设备的访问。对这些地址的访问从逻辑上看，就是内存访问。

读者应该还记得本书第 11 章里面讨论操作系统和用户程序在内存里面的相对位置时，有一种模式是操作系统处于内存地址的两端，用户程序处于内存空间的中间。在这种模式下，内存顶端的操作系统部分就是内存映射的输入输出，即设备的控制寄存器的地址在可用物理内存地址的上方。

内存映射的 I/O 从 DEC 公司的 PDP-11 开始引入。

内存映射虽然具有 I/O 与内存访问统一的优点，但是也存在诸多问题。

第 1 个问题是缓存使用而产生的问题。由于使用内存映射，I/O 控制寄存器被当做内存来看待，而内存里面的内容则有可能缓存起来。对于一般的数据来说，缓存提供的是优势。但对于控制寄存器来说，则是一个大缺点。因为操作系统在读控制寄存器时需要的是最新的内容，而不是缓存的内容。I/O 设备的任何变化都将在控制寄存器内容上体现。但这个内容却不一定在缓存体现。这样，如果读操作在缓存命中，操作系统获得的系统状态将是以前的状态，这将直接影响 I/O 操作的执行。

内存映射的第 2 个问题是总线竞争。在单总线系统里，内存和设备均需要对总线上的数据进行监听，以确认命令是否针对自己。这样将产生总线竞争而降低系统效率，如图 19-3a 所示。如果是多总线系统，则 I/O 设备有可能看不到总线上的地址，如图 19-3b 所示。

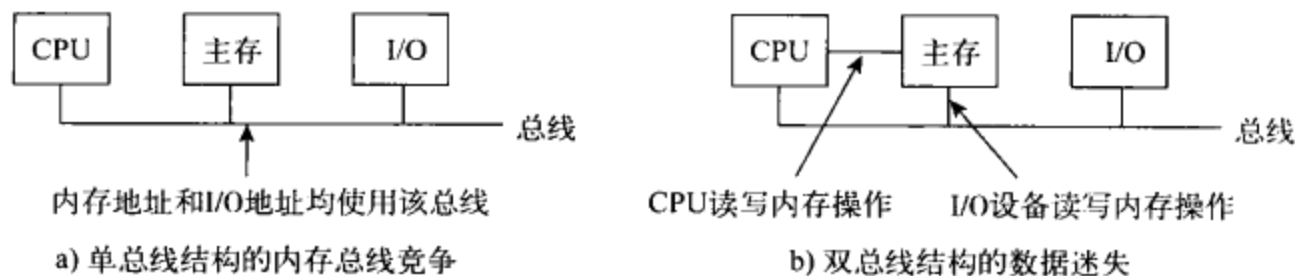


图 19-3 总线和多总线的内存映射问题

对于内存映射的问题，我们也有一定的应对办法。对第1个问题，我们可以使用缓存禁止位。对于多总线数据迷失问题，解决的办法还有多个：

- 失败与再试：如果一个请求没有得到内存响应，则将数据发到另外的总线。
- 窃听：在总线上安装一个窃听装置，负责对数据进行分发。
- 地址过滤：使用一个过滤装置把地址自动过滤到合适的地方。

例如，奔腾处理器结构采用的就是多总线地址过滤模式，如图19-4所示。

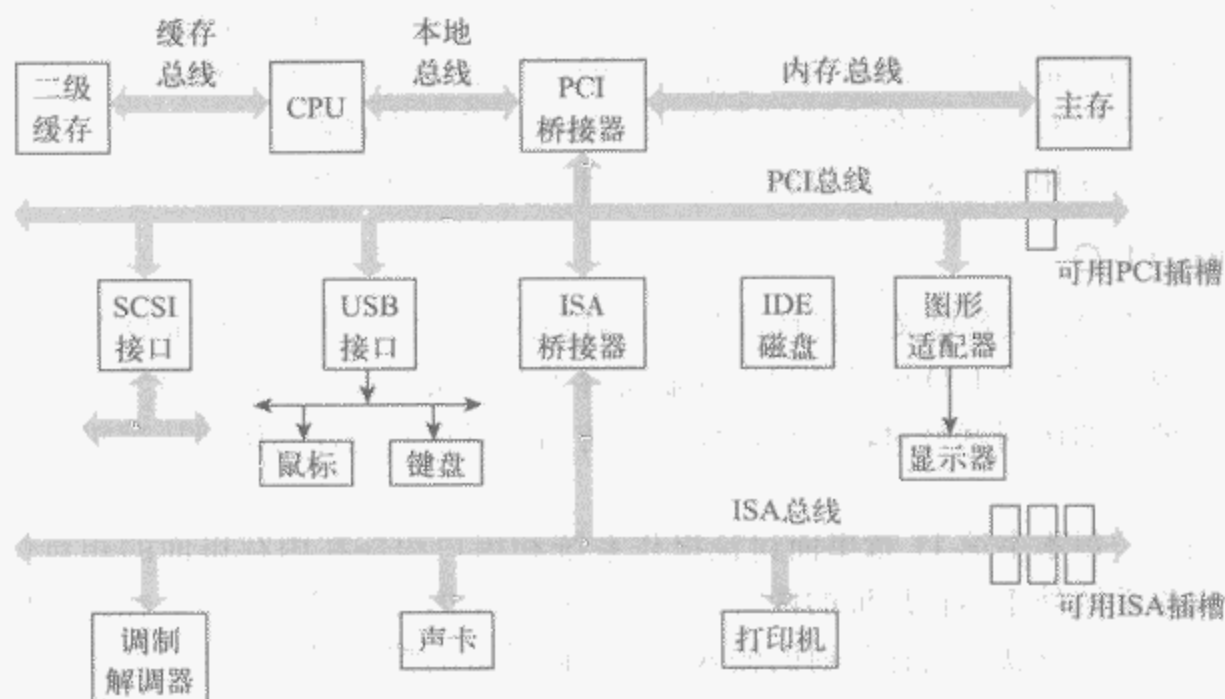


图 19-4 奔腾处理器的地址过滤结构

但对于单总线竞争问题，尚无有效的解决办法。

3. 复合 I/O 模式

当然也可以使用上述两种方式的组合，即复合 I/O 模式。在这种复合模式下，数据缓冲区为内存映射，但是输入输出端口为分开的部分。即设备控制器寄存器需要使用特殊命令来访问。例如，英特尔公司的奔腾处理器将 640K 到 1M 的地址留给设备数据缓冲区，而 0 到 64K 则留给 I/O 端口。

图 19-5 显示了以上三种模式的结构图。

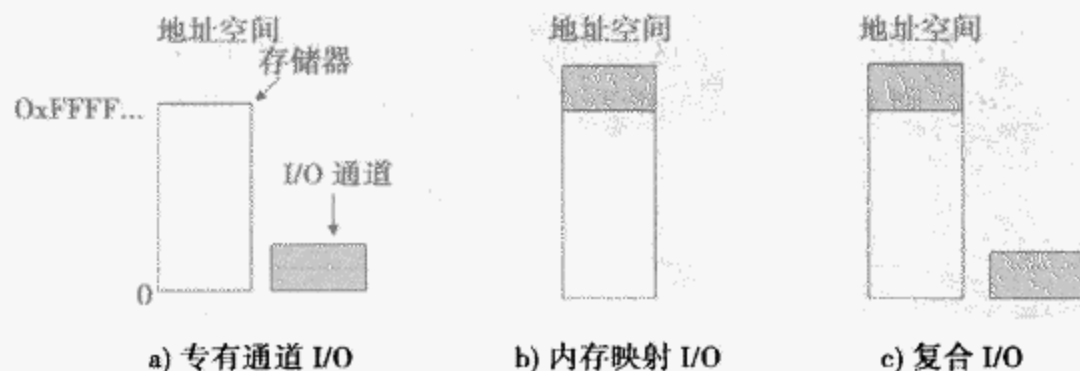


图 19-5 硬件 I/O 模式

19.3.4 根据 CPU 在 I/O 过程中的涉入程度进行分类

按照 CPU 在 I/O 过程中的涉入程度来分类，物理 I/O 模式可以分为：

- 繁忙等待访问
- 直接内存访问

1. 繁忙等待访问

不管是否使用内存映射的输入输出，处理器均需要与 I/O 控制器和数据缓冲区进行数据交换。而这种交换既可以按字节进行，也可以按数据块进行。如果按字节进行，CPU 当然需要在整个过程中介入，即 CPU 在 I/O 过程中一直处于繁忙状态。

显然，让 CPU 在 I/O 过程中一直保持繁忙不是一件好事情。如果要传输的数据是一片连续的内存空间，则我们就可以把 CPU 从 I/O 中解脱出来。解脱出来的办法就是直接内存访问，(DMA, Direct Memory Access)。

2. DMA 访问

DMA 的工作原理是：如果按数据块进行 I/O，即需要传输大量数据时，就无需 CPU 的介入。在这种情况下，我们可以让 I/O 设备与计算机内存进行直接数据交换。而 CPU 则可以去忙别的事情。这种将 CPU 的介入减少的输入输出模式称为直接内存访问。

问题是，将 CPU 从繁忙等待中解脱出来，难道 DMA 的整个数据读写过程不需要使用处理器的功能吗？当然不是的。数据传输当然使用 CPU，只不过这里使用的 CPU 不是计算机里面所有进程共享的 CPU，而是由另外一个 CPU 来负责数据传输。这个另外的 CPU 就是 DMA 控制器。

也许读者会问，这有什么意思，还是需要 CPU 繁忙等待，只不过换成一个不同的 CPU 来进行繁忙等待。何必这么麻烦呢？不如就让通用 CPU 来处理。这里的关键是 DMA 里面的 CPU 可以比通用 CPU 简单、便宜很多，它只需要能够以不慢于 I/O 设备的速度进行数据读写即可。其他复杂功能，如算数运算、移位、逻辑运算等功能皆可以不要。

DMA 控制器既可以构建在设备控制器里面，也可以作为独立的实体挂在计算机主板上。而以独立形式存在的 DMA 控制器更为常见。使用 DMA 进行 I/O 的工作流程，如图 19-6 所示。

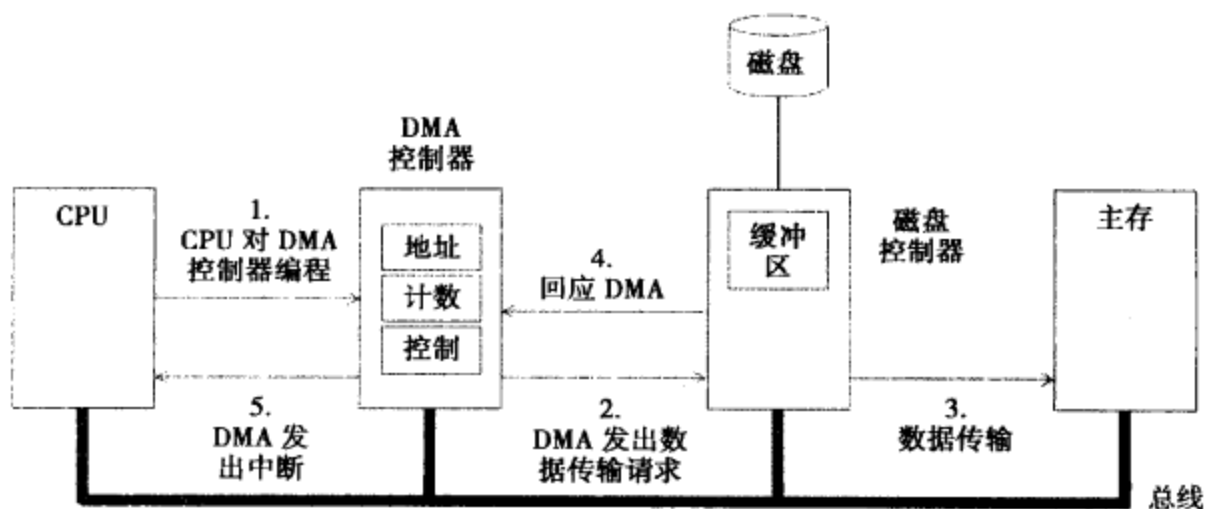


图 19-6 DMA 工作流程

DMA 输入输出的过程如下：

- 1) CPU 对 DMA 进行设置，告诉其 I/O 的起始地址和数据长度。
- 2) 启动 DMA 过程。
- 3) DMA 进行数据传输。
- 4) DMA 结束后发出中断。
- 5) CPU 响应中断并处理结束事宜。

DMA 控制器依不同生产厂商的不同而有很大的不同。最简单的 DAM 控制器在一个时间只能处理一个 I/O，即不能并发；复杂的 DMA 控制器可以同时处理多个 I/O，即它能够提供多个 I/O 通道，每个通道可以对付一个 I/O 设备。

3. DMA 模式的考虑因素

使用 DMA 模式进行输入输出时有一些因素需要考虑：

- 如何访问总线。
- 数据存放何处。
- 内存寻址模式。

由于 DMA 需要使用内存总线，而 CPU 也需要使用内存总线，这样将形成内存总线竞争。那么 DMA 使用总线的模式将依赖于 CPU 使用内存的模式。而根据这些模式的不同，我们可以选择周期盗用（cycle stealing）或者爆发模式（burst mode）。

由于 DMA 数据传输的数据量通常较大，数据存放何处就是一个需要考虑的因素。我们既可以将数据直接存入内存，也可以存入 DMA 缓冲区，然后通过中断让 CPU 一次性将 DMA 缓冲区的数据拷入内存。

另外一个考虑因素就是如何寻址内存。是使用虚拟地址还是使用物理地址？使用物理地址的好处是速度快，缺点是绕过 MMU 存在安全和可靠性风险。

4. DMA 模式的优缺点

优点就是将主 CPU 从 I/O 中解脱出来。而缺点自然是增加了成本和复杂性。而且，由于 DMA 需要与 CPU 竞争内存总线，其效率的提高不如理论上的期望。因此，并不是任何 I/O 都应该使用 DMA。

在下述情况下，应该选择使用主 CPU：

- 设备输入输出速度非常高。
- 主 CPU 没有其他事情可做。
- 成本考虑。

19.4 输入输出软件

前面几节我们从硬件的角度介绍了输入输出，但是光有硬件是不够的。毕竟，对于用户来说，直接对硬件进行操作十分困难。

首先，硬件的种类和类型实在太多，光磁盘就有不同类型、不同生产厂商的数十种之多，而每种磁盘的控制界面并不一样。至于网卡、视频卡、键盘、鼠标等也存在五花八门的品牌和系列。

其次，就是对一种硬件设备来说，其访问过程并不简单。要想正常访问，必须知道该设备的物理结构和特性，其工作原理和机制。而这对于程序设计员来说，显然有点要求过多。

例如，由于对控制寄存器的访问和对内存的访问不同，用户必须明了 I/O 设备的底层命令，并且对于不同的设备这个命令是不同的。这样用户写的程序必须依设备的不同而有所不同，这显然不是一件令人高兴的事情。例如，用户在读写一个文件时，因为介质是软盘、磁盘或光盘，他们需要写出三个版本的程序。

这个问题的答案当然是软件，或者说操作系统软件。我们已经多次强调过，操作系统的角色就是魔幻和管理。在这里也不例外。I/O 软件的目的就是魔幻和管理。魔幻是将不同 I/O 设备的差异屏蔽，使它们看上去似乎是一样的东西，都具有令人爽心悦目的界面；而管理自然是对这些设备进行管理，该独享的独享，该共用的共用，需要缓冲的缓冲，并对设备进行实际的驱动（发出读写命令）。

19.4.1 I/O 软件的目的

从最高层来说，I/O 软件的目的我们已经说过，就是魔幻和管理。而具体来说，其目标有如下几个：

- 设备独立
- 统一命名
- 错误处理
- 数据传输
- 缓冲
- 共用与独享

设备独立指的自然是程序对 I/O 设备的访问不依赖于设备的物理特征，且在输入输出程序的编写时无需事先指定 I/O 设备。统一命名指的是设备或文件的命名不依赖于具体的计算机，这样使用名字将使程序可以在任何机器上运行；错误处理指的是对输入输出过程中产生的数据错误进行侦测与纠正，而且纠错应该在最靠近硬件的层面上进行。数据传输指的是实际操控数据在主机和外设之间的传递，例如支持同步（阻塞传输）和异步（中断驱动）数据传输。缓冲是为数据传输提供一个临时存放地，然后在方便的时候将数据拷贝到最后目的地。共用与独享指的是将设备尽量变为共享，以增大资源利用率和降低死锁发生的概率。例如，将磁盘、打印机变为共享。

19.4.2 逻辑 I/O 模式

在从硬件角度讨论 I/O 时，我们说过可以有不同的 I/O 模式。那么从软件角度来看 I/O，也存在不同的模式，而且这些模式与硬件 I/O 模式遥相呼应。

从 CPU 的涉入程度来分，可以分为可编程 I/O 和中断驱动 I/O，这两种分别对应硬件原理

的繁忙等待 I/O 和直接内存访问。

1. 可编程 I/O 原理

在可编程模式下，CPU 等待 I/O 的完成，即 CPU 涉入程度很深。这种模式也称为轮询，或者繁忙等待。图 19-7 给出的是打印一串字符的可编程 I/O 程序片段。

```
Copy_from_user(buffer,p,count)          /* p 为内核缓冲区 */
For (i=0;i<count;i++) {
    while(*printer_status_reg!=READY);
    *printer_data_register=p[i];          /* 输出一个字符 */
}
Return_to_user();
```

图 19-7 可编程 I/O 举例

显然，在等待打印机状态变为就绪的过程中，CPU 不能干任何事情。而每次等待后做的事情只不过是发送一个字符到打印机的数据寄存器里面。这对于 CPU 来说，真有点“大材小用”了。而且，我们前面说过，大材小用或者浪费资源并不是繁忙等待的唯一甚至最严重的缺陷。最严重的缺陷还有可能造成优先级倒挂，进而造成死锁。

解决的方法就是不繁忙等待，而是使用中断。

2. 中断驱动 I/O 原理

中断驱动就是将 CPU 从繁忙等待中解脱出来。在发送好一个或一批数据后，CPU 就去忙别的事情。I/O 设备处理完这批数据后，向 CPU 发出中断。CPU 响应中断后再发送下一批数据。具体来说，中断驱动过程如下：

- 1) CPU 初始化 I/O 并启动第一次 I/O 操作。
- 2) CPU 去忙别的事情。
- 3) 当 I/O 完成时，CPU 将被中断。
- 4) CPU 处理中断。
- 5) CPU 恢复被中断的程序。

图 19-8 描述的是中断驱动 I/O 里面的后面三个步骤。

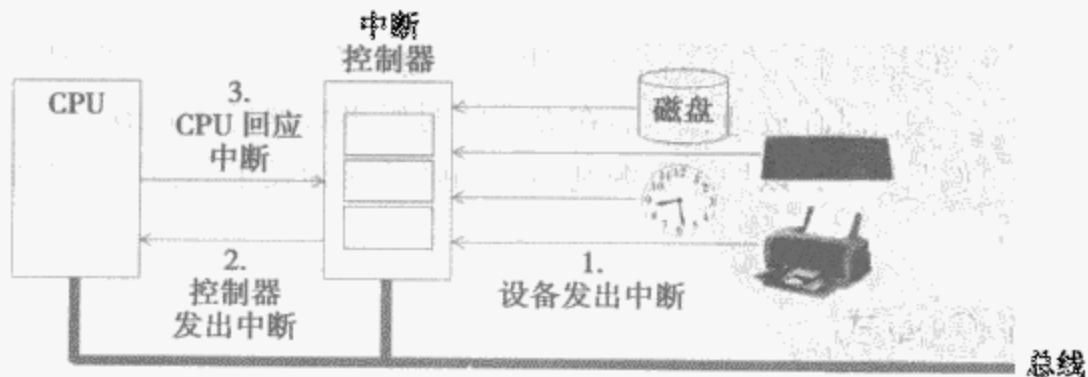


图 19-8 中断驱动 I/O 工作流程

图 19-9 和图 19-10 描述的是中断驱动 I/O 的打印程序片段举例。图 19-9 给出的是打印系统调用时执行的代码，图 19-10 给出的是 CPU 中断响应部分。系统调用片段对应上面所列步骤

的前面两个步骤，中断响应部分对应后面三个步骤。在系统调用片段，我们首先将需要打印的数据从用户空间复制到内核空间，然后启用中断（如果中断本来就是启动状态，这个操作也不会产生任何副作用）。在此之后，等待打印机状态变为就绪，然后发生第一个字符。之后在打印机处理这个字符的输出时，操作系统调用调度器来选取另一个程序来执行。

在中断响应片段，首先检查是否打印完毕，如果是则将用户叫醒。否则，发生下面一个要打印的字符，将仍然需要打印字符数减一。然后回应中断，并从中断处理程序返回（到被中断的程序）。

```
Copy_from_user(buffer,p,count)
Enable_interrupts();
while (* printer_status_reg! =
READY);
* printer_data_register=p[0];
Scheduler();
```

图 19-9 打印系统调用时执行的代码

```
If(count == 0){
    unblock_user();
} else {
    * printer_data_register=p[i];
    Count = count -1;
    i = i +1;
}
acknowledge_interrupt();
return_from_intertupt();
```

图 19-10 CPU 中断响应部分

这里请读者注意，在系统调用部分的初始设置时，操作系统采用轮询等待打印机变为就绪，而不是使用中断。这是因为，操作系统还没有交给打印机任何数据进行打印。

3. 直接内存访问 I/O 原理

中断驱动的 I/O 由于在 I/O 设备工作时无需繁忙等待，其效率比可编程 I/O 要高。但是它毕竟需要 CPU 周期性地中断来发送或接收后续的数据。这种频繁中断对于系统效率来说并不是什么好事。例如，在图 19-9 和图 19-10 的例子中，每一个字符输出均需要中断一次。这样，虽然较可编程 I/O 效率高，但总体来说还是有很大的改进余地。那么怎么改进呢？

当然是降低 CPU 响应中断的频率：从一个 I/O 任务多次中断变为一个 I/O 只有一次中断。这就是直接内存访问。我们在硬件角度看 I/O 时已经讲过了 DMA 的原理。现在从软件的角度看就更加清楚了。由于一个 I/O 任务不一定可以一次将数据传输完毕，因此中间还是需要 CPU 来进行处理，但为了将主 CPU 从 I/O 的繁琐事情中解脱出来，我们另外设立一个 CPU 来响应一个任务中间的中断。而这个 CPU 就是 DMA 控制器。

更进一步来看，既然有一个单独的 CPU 来处理 I/O，也就没有必要使用中断。就让这个额外的 CPU 轮询也不会降低系统效率。因此 DMA 模式下的 I/O 通常是轮询。图 19-11 给出的是使用 DMA 打印时的程序片段。

```
Copy_from_user(buffer,p,count);
Set_up_DMA_controller();
Scheduler();
```

a) DMA 设置部分

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

b) DMA 结束部分

图 19-11 使用 DMA 进行打印时的程序片段

图 19-11a 是 DMA 设置部分。该部分首先将需要打印的数据从用户空间拷贝到内核空间，然后设置 DMA 控制器，包括起始地址和数据量，并启动 DMA。然后调用调度器来选择另一个程序执行。

图 19-11b 是 DMA 结束部分。DMA 结束后会向主 CPU 发出中断。一旦收到这个中断请求，CPU 可以立即回应，而无需检查 I/O 任务是否完成。因为 DMA 模式下，DMA 控制器只有在 I/O 完毕时才会发出中断。回应后叫醒用户，然后从中断响应程序返回。

19.5 I/O 软件分层

从前面所述可以看出，输入输出是一件颇为繁琐的事情。它牵涉到用户空间和内核空间的数据交换、I/O 设备的设置与启动、中断响应与返回，而且整个 I/O 需要提供一个与 I/O 设备无关的统一界面。为了完成一个繁琐的工作，人们通常会将其分为更小的任务来处理。在 I/O 软件上自然不例外。

I/O 软件通常按照 I/O 功能进行分层，每一层提供独特的功能，并与相邻的层面间设计有标准界面。当然，不同的操作系统这种分层也是不同的。但一般，都会有图 19-12 中的几层。

下面我们对每一层的功能进行解说。

- 用户层 I/O 软件
- 设备独立的 OS 软件
- 设备驱动程序
- 中断服务程序

图 19-12 I/O 软件分层

19.5.1 中断服务程序

由于大多数 I/O 均为中断驱动，中断驱动服务程序就成为绝大部分 I/O 软件的不可分割的部分。中断服务程序由于直接与硬件相关，针对不同的 I/O 硬件，中断响应的处理也不尽相同。因此，中断服务程序是 I/O 软件系统分层里面的最底层。而为了降低操作系统的复杂性，中断服务程序的暴露窗口应该越小越好，与其打交道的 OS 部分也是越小越好。

而降低暴露窗口的最好办法是让设备驱动程序负责中断响应。即设备驱动程序启动 I/O 操作后阻塞，然后等待中断。而一个设备驱动程序可以通过操作信号量或睡觉来进行阻塞。当收到中断请求后，中断服务程序先执行，然后将处于阻塞状态的设备驱动程序解锁。这种解锁可以发送信号或者对信号量进行操作。下面为中断处理的步骤：

- 1) 保存没有被中断硬件保存的相关寄存器。
- 2) 设置中断服务程序的上下文。
- 3) 设置中断服务程序的栈。
- 4) 回应中断控制器。
- 5) 重开中断。
- 6) 从保存处恢复寄存器。
- 7) 执行服务程序。
- 8) 设置 MMU 以执行下一个进程。

- 9) 设置新进程的寄存器。
- 10) 开始执行新进程。

19.5.2 设备驱动程序

设备驱动程序，顾名思义，就是直接驱动 I/O 设备进行输入输出操作的软件。它属于与设备控制器直接联系的 I/O 软件部分。它与具体的 I/O 设备直接相关，并针对每个特定的 I/O 设备进行过优化。设备控制程序通常由设备制造商提供，但归属于操作系统内核。正因为这一属性，设备驱动程序是操作系统安全的一个巨大隐患。

由于需要直接驱动设备的运行，设备驱动程序必须清楚设备的所有细节。例如，磁盘驱动程序必须清楚什么是扇面、磁道、磁柱、磁头、磁臂、电机、电机驱动等信息；鼠标驱动程序自然需要具备辨认是哪个键被按下所具有的能力。

设备驱动程序需要完成任务包括：

- 从上层接收抽象的读写请求。
- 确保读写操作被完成。
- 初始化设备，开、关设备。
- 对设备的功能进行管理。

多数操作系统都定义一个标准的块设备界面和字符设备界面。每种界面由一组子程序组成。这组子程序可以由操作系统的其他部分调用。

在 UNIX 系统下，设备驱动程序与整个 OS 编在一个二进制文件里。如果要增加新的设备驱动程序或者修改现有的设备驱动程序，则需要重新编译操作系统。UNIX 的这种做法主要是考虑到安全的因素。由于用户或其他人无法动态地加载设备驱动程序，操作系统安全性较高。不过，这种不能动态加载的限制让人感觉十分不便，因此在最新的 UNIX 操作系统如 Solaris 10 里，设备驱动程序可以动态加载。

Windows 操作系统从一开始（NT 开始）就运行设备驱动程序进行动态加载。这也是 Windows 不如 UNIX 安全的一个原因。

19.5.3 设备驱动程序操作

在收到一个 I/O 请求后，设备驱动程序做的第一件事情是检查输入参数是否合法。如果不合法，则返回错误。如果参数正常，则将 I/O 请求的抽象表示转换为设备能够认识的具体表示，如将线性数据块号转换为磁头、磁道、扇面等。然后，设备驱动程序需要检查设备状态以确认其是否处于闲置状态。如果设备繁忙，则将 I/O 请求送入该设备的等待队列里面以待处理。如果设备没有工作，则驱动设备运行并启动电机。

在做完上述各项工作后，剩下的就是真正的 I/O 操作了。设备驱动程序通过发出一系列的输入输出命令到设备控制寄存器里面来进行真正的数据传输工作。如果需要，设备驱动程序需要阻塞并等待中断。而每次从中断醒过来，则需要检查 I/O 是否正确完成，再将数据传输到上面调用设备驱动程序的应用。

在所有数据传输完毕后，设备驱动程序将返回到调用者那里。

19.5.4 设备独立的操作系统软件

一般来说,设备驱动程序并不直接从用户处接受 I/O 请求,而是通过另外一层中介获得用户请求。这层介于设备驱动程序与用户程序之间的中介就是设备独立的操作系统软件。操作系统在设计时之所以有这层软件是因为 I/O 软件的一部分与设备有关,一部分与设备无关。而如果与设备无关,就可以将这部分共用起来,放置在设备驱动程序之上,为用户提供一个统一的 I/O 界面。

这种对于所有 I/O 设备都一样的操作包括诸如缓冲、错误报告、分配与释放独享设备、提供设备独立的数据块尺寸等。设备独立的 OS 软件与设备驱动软件之间的分界自然与每个设备有关。

1. 统一界面

设备独立的操作系统软件(见图 19-13)的一个重要目标是提供一个统一的 I/O 界面。即让所有的 I/O 设备看上去一样或者相似,这是操作系统经常扮演的角色。使用的办法则是将设备驱动界面进行标准化:规定一个设备驱动程序必须提供的功能清单;规定内核为设备驱动程序提供的功能清单和界面。

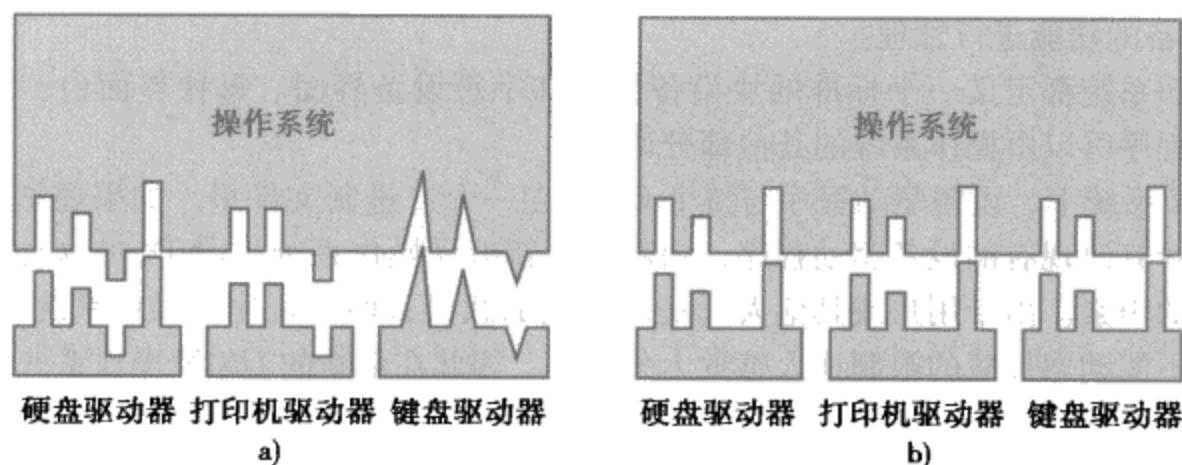


图 19-13 设备独立的 OS 软件通过标准化设备驱动程序
界面提供统一 I/O 界面(来源:参考文献 [3])

2. 缓冲

缓冲是几乎所有 I/O 设备都需要的一种操作。缓冲的目的有两个:一是桥接速度不同的设备,使之可以沟通同步;二是提供灵活的健壮机制,因为在每个缓冲层都可以进行一些健壮性、可靠性、安全性处理。而第一点又有两层意思,一是提高数据传输速度,因为快速设备不必等待慢速设备;二是防止溢出,因为慢速设备来不及处理的数据可以存放在缓冲区而不会丢失。

当然,缓冲也有缺点:就是降低了数据传输的时效性。因为数据层层缓冲处理是需要时间的。如果一个系统的时效性非常重要,则最好不要使用缓冲,而是在通信双方之间创建一个没有缓冲的直接通道,这样,数据从一方发出后,另一方将马上收到。例如, Mach 操作系统的 x-kernel 就提供此种无缓冲的直接数据通道。自然,这种行为是比较危险的行为,必须非常小心来避免数据溢出和丢失。

3. 错误报告

在 I/O 操作中，错误是难免的。这是因为 I/O 需要与计算机外面的世界打交道。而外面的世界自然不如计算机内部的世界井然有序。统计数据表明，计算机壳里面发生数据传输错误的概率很低，而在计算机壳外部进行数据传输发生错误的概率相当高。二者可以相差几百万倍。因此，在 I/O 中进行错误处理是件十分重要的事情。

如何进行错误处理取决于错误的类型。一般来说，错误可以分为程序错误和真正的 I/O 错误。程序错误就是用户要求设备做一件该设备无法做到的事情，例如从输出设备上读数据。真正的 I/O 错误当然是指数据读写过程中发生的错误，例如数据读错了，或者磁盘盘片损坏了等。

对于程序设计错误来说，I/O 软件除了将错误报告给用户外，似乎不能做任何别的事情了。但如果是 I/O 错误，那 I/O 软件则需要进行适当的纠错操作，看能否消除错误。如果不能消除错误，可以询问调用者如何处理，或者干脆返回一个错误码给用户。

19.5.5 用户层 I/O 软件

前面说过，设备驱动程序从设备独立的操作系统软件层接受 I/O 请求。而设备独立的操作系统软件则从用户或应用软件处接受指令。但是用户也好，应用软件也好，如果需要向 I/O 软件发出指令，得需要一个发出指令的界面。这个用户层可以直接使用的 I/O 界面就是用户层 I/O 软件。这一层既然可由用户直接操控，它自然运行在用户空间。

例如，很多读者都见过的 `Count = write (fd, buffer, nbytes)` 命令就是用户层 I/O 软件的一部分。这条指令里面的 `write` 被很多人误认为是操作系统的系统调用，而实际上并不是。这是一个由高级语言提供的库函数，它将操作系统的相关（在这个例子里是相同的）系统调用包裹起来。用户与这个库函数打交道，而这个库函数在编译出来后，会变成一系列指令，来完成系统调用过程。

除此之外，放在用户层 I/O 软件里面的 I/O 功能包括格式化、假脱机等。

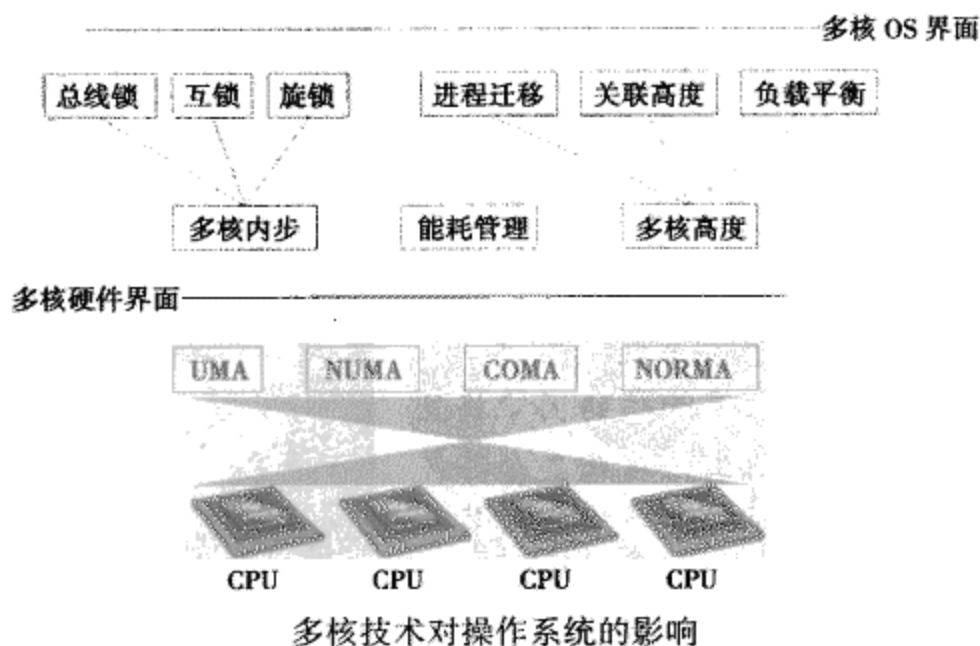
思考题

1. 从硬件视角看，有哪些输入输出模式？
2. 从软件角度看，有哪些输入输出模式？
3. 哪些输入输出模式需要使用轮询？
4. 将抽象 I/O 命令转换为具体 I/O 指令是什么部分的责任？
5. 缓冲在输入输出中起的是何种作用？
6. 直接内存访问是怎么的工作原理？
7. 有人认为直接内存访问成本高，有人认为其 I/O 成本低，你认为如何？为什么？
8. 为什么在 I/O 时进行错误处理非常重要？
9. 设备驱动程序通常由设备制造商，而非操作系统设计者提供。这个事实对操作系统设计产生何种影响？为什么？
10. 中断是操作系统运转的根本基础，请论述中断在 I/O 处理中的重要性。

第六篇 多核原理篇

随着多核技术的流行，操作系统作为计算机的管理者，不得不增加对多核的管理功能。而从单核到多核，并不只是处理器或处理核数量的简单变化，其对操作系统的影响是多方位的。首先，多核的出现对操作系统的进程和线程调度产生了直接影响。因为这个时候可以同时调度多个线程和进程进行执行。其次，在多核环境下，一条指令可以同时多个核上面执行，就使得单核环境下的同步机制不一定运行正确。另外，多核的出现使得能耗的管理趋于重要。

本篇对新出现的多核技术进行讲解。重点讨论多核环境给操作系统带来的影响。全篇分为多核结构和多核操作系统两章。第20章的内容包括多核处理器结构（超线程结构、多核结构、多核超线程结构）、多核内存结构（UMA、NUMA、COMA、NORMA）、对称多核处理器计算机的启动过程、多处理器之间的通信、SMP缓存一致性等。第21章的内容包括多核进程同步、多核环境下的软件同步原语、旋锁及其实现、队列旋锁、多核环境下的进程调度、多核环境下的能耗管理和多核系统性能。



本篇最重要的概念是多核之间的协调：即如何让多个核的计算能力相得益彰，而不是相互抵消。从某种程度上说，多核之间的协调与前面讲过的进程或线程之间的协调具有相通性。而从另一个角度看，这种协调又是进程之间或线程之间协调的扩展和延续，因此，只要明白了线程之间的协调原理与机制，理解多核技术及其对操作系统的影响就是水到渠成的事情了。

第 20 章 多核结构与内存

引子

由于人类的贪婪本性，我们对计算机芯片的速度总是感到不满意。每次芯片速度的提升带来的是更加贪得无厌的需求，从而造成人类对芯片速度提升的不断追求。自从 1979 年推出频率为 5 MHz 的 8088 处理器以来，英特尔公司一直在努力提升 CPU 的时钟频率，其最新最快的奔腾芯片时钟频率已经达到了令人咂舌的 3.8 GHz。

但是与程序不同的是，计算机不能违背物理定律。如果芯片需要快速运算，如每几个纳秒进行一次状态变换，则只能使电子传输一个很短的距离，如最多 50 cm。而这就需要很多元件集中在很小的空间区域内，从而产生大量的热量。这些热量必须想办法散出去，否则芯片将被烧毁。在过去，工程师们在此方面进行了卓越的努力。

但随着芯片主频的提升，芯片的散热成为了一个不可逾越的障碍。在单位面积上聚集元件的数量也已经饱和，这使得处理器性能的改善遇到了瓶颈。这个瓶颈打破了几十年来一直保持的芯片时钟频率（又或单位芯片上元件数量）每两年增加一倍（或每 3 年翻两番）的摩尔定律。芯片工业的技术专家早就预言过：随着芯片速度的提升，工程人员面临的困难将呈指数级增加。事实上，芯片生产商在单个芯片的性能上已经无计可施。英特尔多次提到其在攻克 4G 芯片中遇到了各种难以克服的困难。而难中之难就是芯片产生的热量令人窒息。

借用英特尔技术总监帕特里克·格辛格的话说：“如果芯片设计不出现根本性变革，十年内，运行在更高速度的个人机芯片将变得如太阳般炙热（见图 20-1）。”

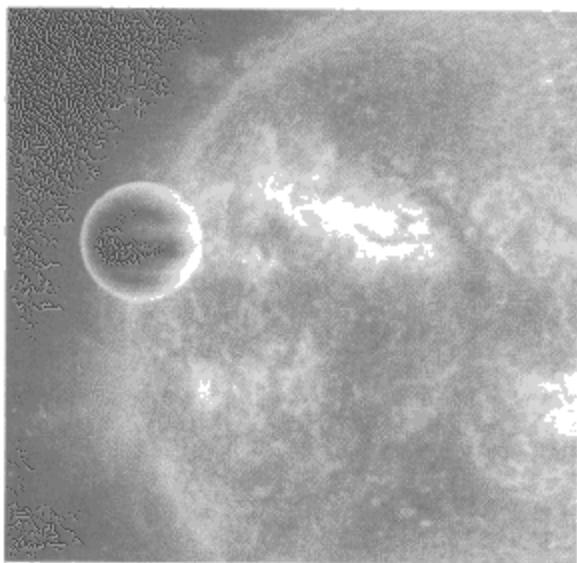


图 20-1 在传统技术下，运行在更高速度下的未来芯片将变得如太阳般炙热

20.1 以量取胜

明知芯片速度的提升已经达到难以为继的境界，但是人类对速度的追求却并没有丝毫停歇的意思。那怎样在不烧毁计算机的情况下满足人类漫无止境的贪婪呢？

质量上不行，数量上补。这是人类几千年来总结出的“金科玉律”。既然无法将单块的CPU速度提升到令人满意的境界，那就使用多个CPU来满足人类的贪欲。也就是说，如果技术上打不过对手，我们就以数量取胜！而以多取胜的核心是就是本章将要论述的多核技术。

在今天，芯片厂商已经将他们的将来压宝在多处理器和多核上面。多处理器指的是在一个体系结构上放置多个CPU，而多核则指在同一块芯片（CPU）上放置多个核（core），即执行单元。多核和多CPU的区别是多核结构更加紧凑，成本在同等执行单元数量的情况下更便宜、功耗更低。例如，具有两个执行单元的双核处理器就比使用两个处理器的多处理器结构便宜、紧凑、功耗低。为简单起见，人们现在将多处理器和多核结构统称为多核结构。

多核技术的开端是所谓的双核。该概念最早由IBM、HP、Sun等支持RISC架构的高端服务器厂商提出，主要运用于服务器上，如IBM于2001年推出的双核RISC芯片Power 4，HP于2004年2月推出的PA-RISC8800，和Sun于2004年3月推出的UltraSPARC IV的双内核处理器。但让多核成为家喻户晓的技术名词则是在多核进入到IA阵营后。在AMD和Intel分别将多核引入到个人PC机后，多核技术迅速得到普及。随着AMD和Intel在多核技术上的大力研究和推进，多核技术已经从双核推进到4核、8核甚至更高。而基于多核技术的计算机产品也已经比比皆是。芯片厂商期待这种多核结构能够改善整块芯片的处理能力（因为有多处理器），提升芯片的吞吐量，从而达到间接提升芯片性能的目的。

多核计算机的出现，打破了单核环境下的许多操作系统设计的正确性或可靠性。例如，我们在锁的实现时讲过的以中断启用和禁止来实现锁的做法，在多核环境下将不能工作。事实上，本书前面两大部分讲述的进程和内存管理的许多策略和机制针对的均是单CPU或单核的环境。这些策略和机制在多核环境下要么是不能正确运行，要么是效率太低，当然也有可能是二者兼而有之。因此，为了适应多核环境所提出的新要求，也为了更好地利用多核技术提供的新方便，操作系统需要作出相应调整。本章就对多核环境对操作系统的影响进行讨论。

本书先对多核的体系结构做一简单介绍，然后讨论多核环境下的进程和内存管理。由于文件系统基本不受多核环境的影响，我们将不予讨论。

20.2 多核基本概念

要讨论多核环境下的操作系统所做的调整，首先需要知道多核环境和单核环境的不同之处在什么地方。我们首先来看一下多核的一些基本概念。在x86体系结构下，多处理功能芯片经过了对称多处理器结构、超线程结构、多核结构和多核超线程结构的4个演变阶段。我们下面分别予以介绍。

20.2.1 多处理器结构

除了提升 CPU 主频和增加一、二级缓存容量外，提升计算机性能的最直截了当的办法就是在—台电脑里面安装多个 CPU。由于 CPU 个数增加，电脑同时处理的工作量就增加，自然提升了系统吞吐量和改善了用户响应时间，从而感觉到计算机的性能得到了提升。

多处理器结构说简单一点就是在—条总线上挂载多个处理器。在传统的体系结构下，—台电脑里面只有—个 CPU。而在多处理器系统里，—台电脑里面可以有多个 CPU。图 20-2 给出的就是英特尔公司的有两个 CPU 的计算机体系结构。

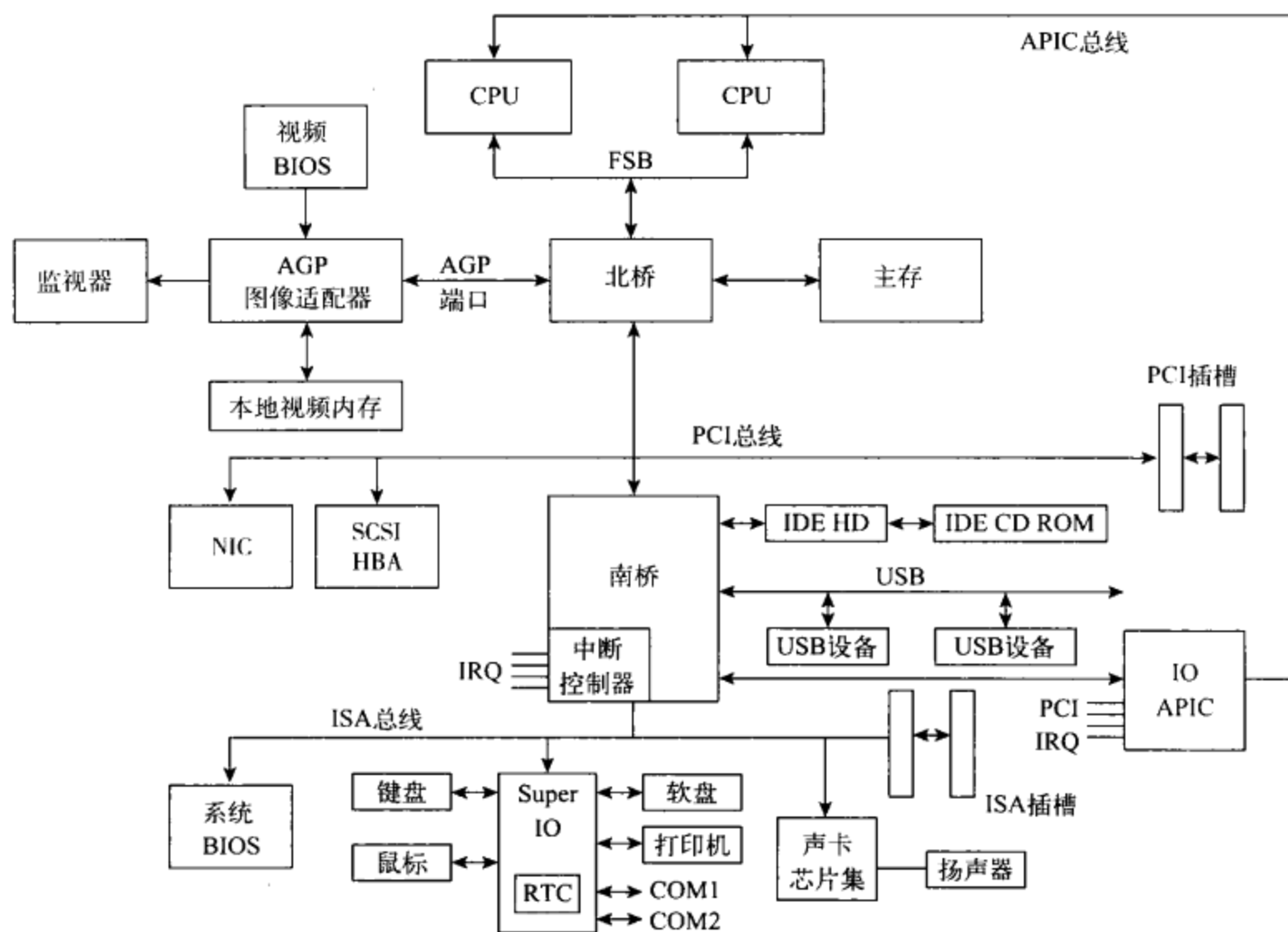


图 20-2 多处理器结构

在多个 CPU 的情况下，以 CPU 之间的关系不同又可以分为对称和非对称多处理器结构。在对称结构下，多个 CPU 的角色功能平等，没有主从之分，这种多 CPU 结构称为对称多处理器结构（Symmetric Multi-Processor Architecture），或简称为 SMP 结构。而在非对称多处理器结构下，不同 CPU 的角色地位不同，有所谓的主从 CPU 之分。这种多 CPU 结构称为非对称多处理器结构（Asymmetric Multi-Processor Architecture），或简称为 AMP 结构。由于 SMP 结构远比 AMP 结构普遍，本章讨论的多处理器结构将只针对 SMP。

当然，我们并不需要限制在两个 CPU 上。我们也可以在—台电脑里面安装多于两个的

CPU。图 20-3 给出的就是一台有着 4 个 CPU 的计算机体系结构简化图。

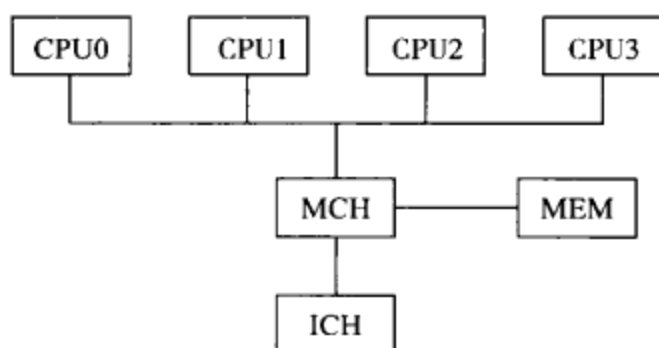


图 20-3 有着 4 个物理 CPU 的对称多处理器结构

20.2.2 超线程结构

在一台电脑里安装多个 CPU 虽然提升了计算机的性能，但是付出的代价是高昂的成本和巨大的功耗。而在实际中，基于很多原因，CPU 的执行单元并没有被充分使用。如果 CPU 不能正常读取数据（总线/内存的瓶颈），其执行单元利用率会明显下降。另外就是目前大多数执行线程缺乏 ILP（Instruction-Level Parallelism，多种指令同时执行）支持。这些都造成了目前 CPU 的性能没有得到全部的发挥。因此，Intel 提出了超线程（Hyper Threading）技术来让一个 CPU 同时执行多重线程，从而提高 CPU 效率和用户满意度。

超线程技术是在一个 CPU 上同时执行的多个程序共同分享该 CPU 内的资源，理论上像两个 CPU 在同一时间执行两个线程。超线程技术可在同一时间里，让应用程序使用芯片的不同部分。而为了支持这种技术，需要在处理器上多加入一个逻辑处理单元指针（Logical CPU Pointer）。因此新一代的 P4 HT 的模板的面积比以往的 P4 增大了 5%。而其余部分如 ALU、浮点运算单元、二级缓存则保持不变。

图 20-4 描述的是超线程结构。图中的每个 CPU 并不是物理上的单个 CPU，而是两两为一个独立的 CPU。即图中只有 4 个物理 CPU，而每个 CPU 又因超线程技术被分解为两个逻辑 CPU。每个逻辑 CPU 可以执行一个线程序列。这样一个物理 CPU 可以同时执行两个线程。

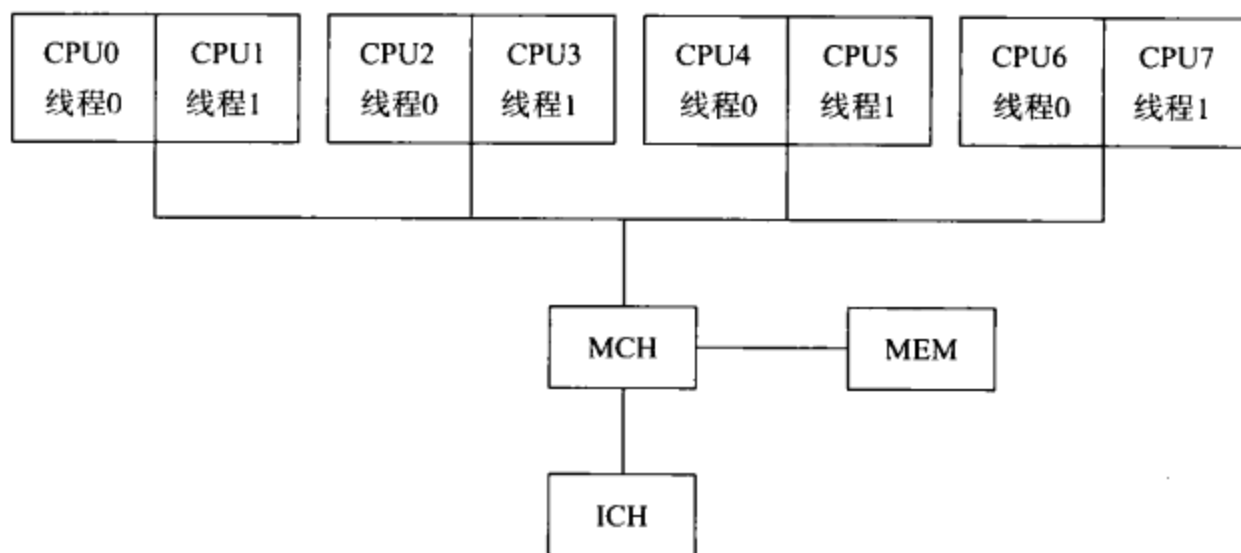


图 20-4 四个物理 CPU、8 个逻辑 CPU 的超线程结构

需要注意的是,含有超线程技术的 CPU 需要芯片组和软件支持,才能比较理想地发挥该项技术的优势。操作系统如 Windows XP、Windows 2003、Linux 2.4.x 以后的版本均支持超线程技术。

虽然采用超线程技术能同时执行两个线程,但它并不像两个真正的 CPU 那样,每个 CPU 都具有独立的资源。当两个线程都同时需要某一个资源时,其中一个要暂时停止,并让出资源,直到这些资源闲置后才能继续。因此超线程的性能并不等同于两个 CPU 的性能。

20.2.3 多核结构

多 CPU 成本高、功耗大,而超线程技术又不等同于两个 CPU 的性能,而且时常会碰到两个线程需要同一资源时必须停止一个线程的现象。

那么有没有什么办法同时克服多 CPU 和超线程的缺点呢?即性能像多 CPU、功耗像超线程的结构呢?有,就是多核结构 (Multi-core Architecture)。

多核结构就是在一个 CPU 里面布置两个执行核,即两套执行单元,如 ALU、FPU 和 L2 缓存等。而其他部分则两个核共享。这样,由于使用的是一个 CPU,其功耗和单 CPU 一样。由于布置了多个核,其指令级并行将是真正的并行,而不是超线程结构的半并行。

例如,英特尔公司的奔腾 D 和奔腾 EE (见图 20-5) 即是分别面向主流市场以及高端市场的双核芯片。其每个核采用独立式缓存设计,在处理器内部两个核之间是互相隔绝的,通过处理器外部 (主板北桥芯片) 的仲裁器负责两个核之间的任务分配以及缓存数据的同步等协调工作。两个核共享前端总线,并依靠前端总线在两个核之间传输缓存同步数据。

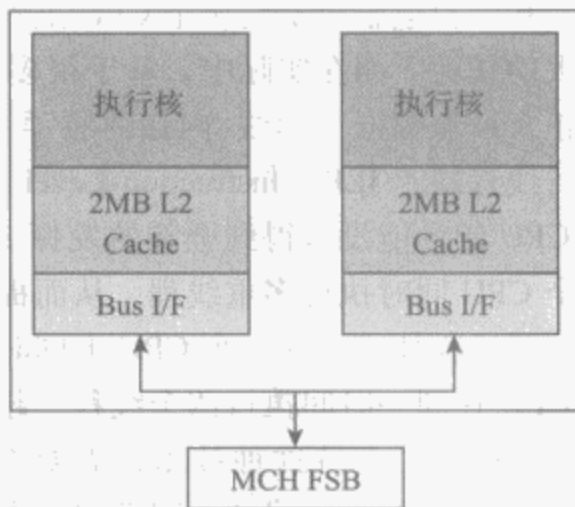


图 20-5 奔腾 D 和奔腾 EE 双核芯片
(来源:英特尔公司)

当然,我们也可以在计算机里面放置多个配置有多个执行核的 CPU,而形成更多的核。

例如,如果我们用 4 个双核 CPU 则可以构建如图 20-6 所示的多核、多处理器体系结构。

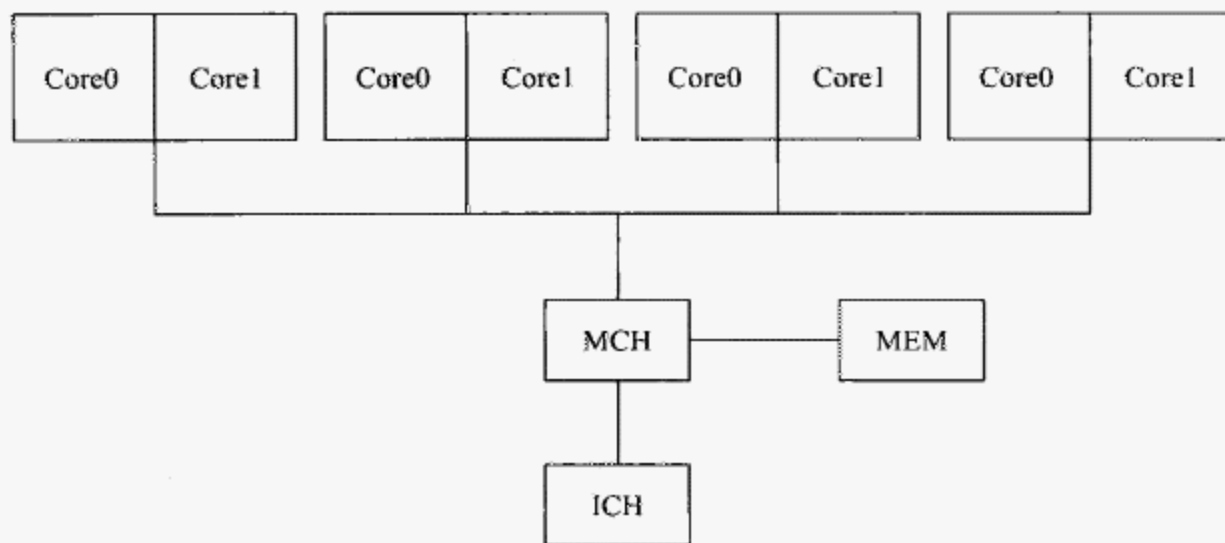


图 20-6 有着 4 个 CPU、8 个核的计算机体系结构

20.2.4 多核超线程结构

而在多核情况下，我们也可以将超线程技术予以使用，从而形成多核超线程（Multi-core Hyper Threading Architecture）技术。即每个物理执行核里面又分解为两个或多个逻辑执行单元，如图 20-7 所示。

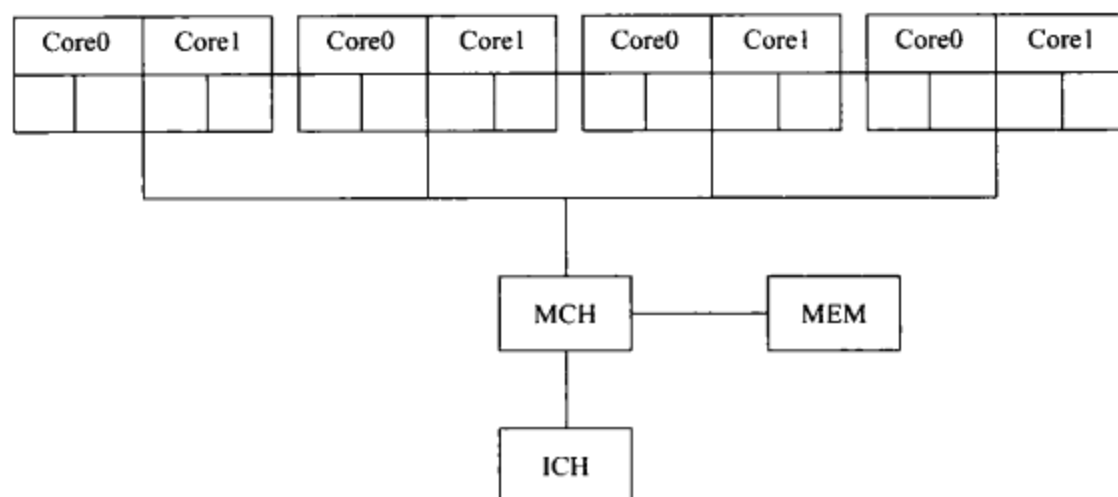


图 20-7 有着 4 个 CPU、8 个核、16 个逻辑执行单元的多核超线程结构

例如，英特尔公司的奔腾 EE 多核芯片就支持超线程技术，一块奔腾 EE 芯片在打开超线程技术之后会被操作系统识别为四个逻辑处理器。

20.3 多核的内存结构

由于一台计算机里面有多个执行核，而每个执行核均需要对内存进行访问，那么这种访问在多个核之间是如何协调的呢？或者说内存在多个核之间是如何分配的呢？

20.3.1 UMA

最简单的内存共享方式就是将内存作为与执行核独立的单元构建在核之外，所有的核通过同一总线对内存进行访问。由于每个核使用相同的方式访问内存，其到内存的延迟也相同，这种访问模式我们称为均匀内存访问，即所谓（UMA，Uniform Memory Access）。在这种模式下，最重要的是所有核的地位在内存面前平等。其优点是设计简单，实现容易。缺点是大锅饭，难以针对个体的程序进行访问优化，和扩展困难。因为随着执行核数的增加，对共享内存的竞争将变得白热化，从而造成系统效率急剧下降。

当前的对称多处理器共享存储系统基本上采用此种模式。这种模式只能在处理器个数或执行核数量较少时方可使用。

20.3.2 NUMA

如果我们想构建 CPU 数量很多的多处理器系统，或者欲构建执行核多于 4 个的多核系统，则 UMA 结构因内存共享瓶颈而不能胜任。在这种情况下，一种自然的选择是使用多个分开的

独立共享内存。每个执行核或 CPU 到达不同共享内存的距离不同，访问延迟也不一样。这种访问延迟不一致的内存共享模式称为非均匀内存访问，即所谓的（NUMA，Non-Uniform Memory Access）。在这种模式下，最重要的特点是执行核在不同的内存单元面前地位并不平等：到近的内存具有优势地位，而到远的内存则属于劣势。

在 NUMA 下，原则上应该将程序调度到离本地内存（程序存放的内存单元）近的执行核上，以提升程序的内存访问效率，从而提高程序的执行效率。

NUMA 结构的优点是灵活性高、扩展容易。在执行核的数量增加的时候，其访问内存的效率可以保持不下降。不过，这种不下降的前提是优良的调度策略，即在调度时能够将程序就近执行；否则，有可能因内存访问距离远而造成效率下降。因此，NUMA 对调度的要求很高。但因为扩展容易，NUMA 得到了非常广泛的应用。

例如，如果想组建更多的核，我们可以将两个图 20-7 的结构组合成为更多核的、多内存的 NUMA 结构，如图 20-8 所示。

例如，英特尔公司就是以 NUMA 方式构建了 80 个核的芯片，如图 20-9 所示。

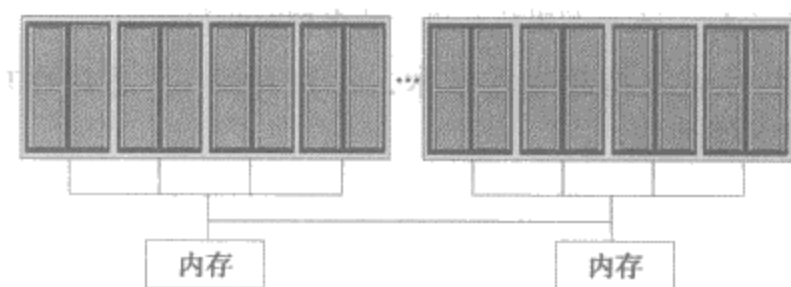


图 20-8 非均匀内存访问的多核多处理器结构

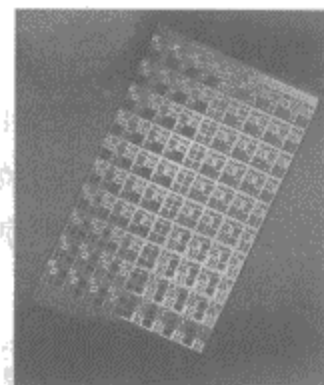


图 20-9 英特尔的 80 核多核芯片
(来源：英特尔公司)

20.3.3 COMA

我们前面说过，NUMA 具有灵活、易扩展的优点，但对调度的要求高。如果不能或难以将程序调度到就近的执行核上，那又怎么办呢？答案是老办法：缓存。即在每个执行核里面配置缓存，其执行需要的数据均缓存在该缓存里面。所有访问由缓存得到满足。这样，不论数据原来是处于哪个内存单元，其对效率的影响均将不复存在。

这种完全由缓存满足数据访问的模式称为全缓存内存访问，即所谓的（COMA，Cache Only Memory Access）。在这种模式下，每个执行核配备的缓存共同组成全局地址空间。

20.3.4 NORMA

如果内存单元为每个执行核所私有，且每个执行核只能访问自己的私有内存，对其他内存单元的访问通过消息传递进行，则就是所谓的非远程内存访问模式，即（NORMA，Non-Remote Memory Access）。这种模式的优点是设计比 NUMA 还要简单，但执行核之间的通信成本高昂。这已经有一点像网络了。因为效率问题，此种模式在多核体系结构下使用甚少。

20.4 对称多处理器计算机的启动过程

对于单处理器或单核计算机来说,其启动过程没有什么悬念。因为只有一个CPU。但对于多核和多处理器来说,问题就不是这么简单了。计算机启动到底意味着什么?是只要有一个CPU启动就算启动,还是所有CPU启动才叫启动?多个CPU有没有启动顺序?

显然,多个CPU不可能同时启动,必定有先后次序之分,因为我们不能让两个CPU同时执行BI/OS(或EFI)里面的指令(BI/OS不支持多线程)。因此,除一个CPU外,必须让其他CPU均处于中断屏蔽状态。也即CPU的启动是有次序的。

问题是这个次序是固定的,还是随机的?

对于对称多处理器结构来说,这个顺序是固定的。所有的CPU里面有一个被定为启动处理器(BootStrap Processor, BSP)。而其他的处理器则作为应用处理器(Application Processor, AP)。而到底哪个CPU是BSP则由某一特定寄存器的值来决定。例如,对于英特尔公司的多处理器结构来说,寄存器IA32_APIC_BASE_MSR里面有一个BSP标志位。该位被设置则意味着该处理器是BSP处理器。而在任何时候只能有一个CPU的BSP标志位被设置。这一点可用共系统总线裁决(arbitration on the system bus)来实现。

有了BSP和AP的区别后,SMP的启动过程就比较清楚了:

- 1) BSP首先读取并执行BIOS(或EFI)的初始化(boot-strap)代码(通常处于物理地址FFFF FFF0H)对自己进行初始化,这种初始化包括设置高级可编程中断控制器(Advanced Programmable Interrupt Controller, APIC)环境,建立全局的数据结构,设置跳转代码(trampoline code)并准备AP的运行环境。

- 2) 而AP则在上电或重启后进行一个简单的自我设置后进入到等待启动状态。

- 3) 然后BSP通过发送进程间中断来叫醒AP,对AP进行启动并令其进行初始化。

- 4) AP在收到BSP发出的IPI启动信号后,则将执行BIOS(或EFI)里面的AP初始化代码或BSP准备的跳转代码。该跳转代码将初始化AP处理器。此后,AP再次进入到等待BSP中断的状态。

- 5) 而BSP则继续执行BIOS或EFI的后续代码,并负责启动操作系统。

值得一提的是,SMP的BIOS与单核或单处理器里的BIOS并不一样。由于有多个处理器,SMP的BIOS里面包括了多个处理器的规格和信息,每个处理器的APIC描述表。而这些信息包括诸如CPU的数量与编号、本地APIC信息、I/O的APIC信息等。由上面的启动过程也可以看出,SMP的BIOS里面不光有BSP的初始化代码,通常还包括AP的初始化代码。

20.5 多处理器之间的通信

既然一个系统里面有多个CPU,这些CPU之间总需要进行某种通信,以进行任务的协调。而这种协调既可能是CPU本身需要,也可能是运行在它们上面的进程和线程之间需要。那么

CPU 之间的通信方式是什么呢?

我们讲进程时讨论过进程间的通信:管道、套接字、信号、信号量、消息队列、共享内存等。这些机制能否用来实现多 CPU 之间的通信呢?

在多 CPU 之间通信,自然也可以发送信号。不过这个信号不是内存的一个对象,因为这样的话,无法及时引起另外一个 CPU 的注意。而要引起其注意,需要发送的是中断。

而用来协调这些 CPU 之间中断的机制就是所谓的高级可编程中断控制器 APIC。这是实现 SMP 功能必不可少的,且是 Intel 多处理规范的核心。在此种规范下,每个 CPU 内部必须内置 APIC 单元(成为那个 CPU 的本地 APIC)。CPU 通过彼此发送中断(所谓的 IPI,处理器间中断)来完成它们之间的通信。通过给中断附加动作(action),不同的 CPU 可以在某种程度上彼此进行控制。

除了每个 CPU 自己本地的 APIC 外,所有 CPU 通常还共享一个 I/O APIC 来处理由 I/O 设备引起的中断,这个 I/O APIC 是安装在主板上的。图 20-10 描述的是英特尔公司的 Xeon 多处理器结构下的本地 APIC 和 I/O APIC 的结构示意图。

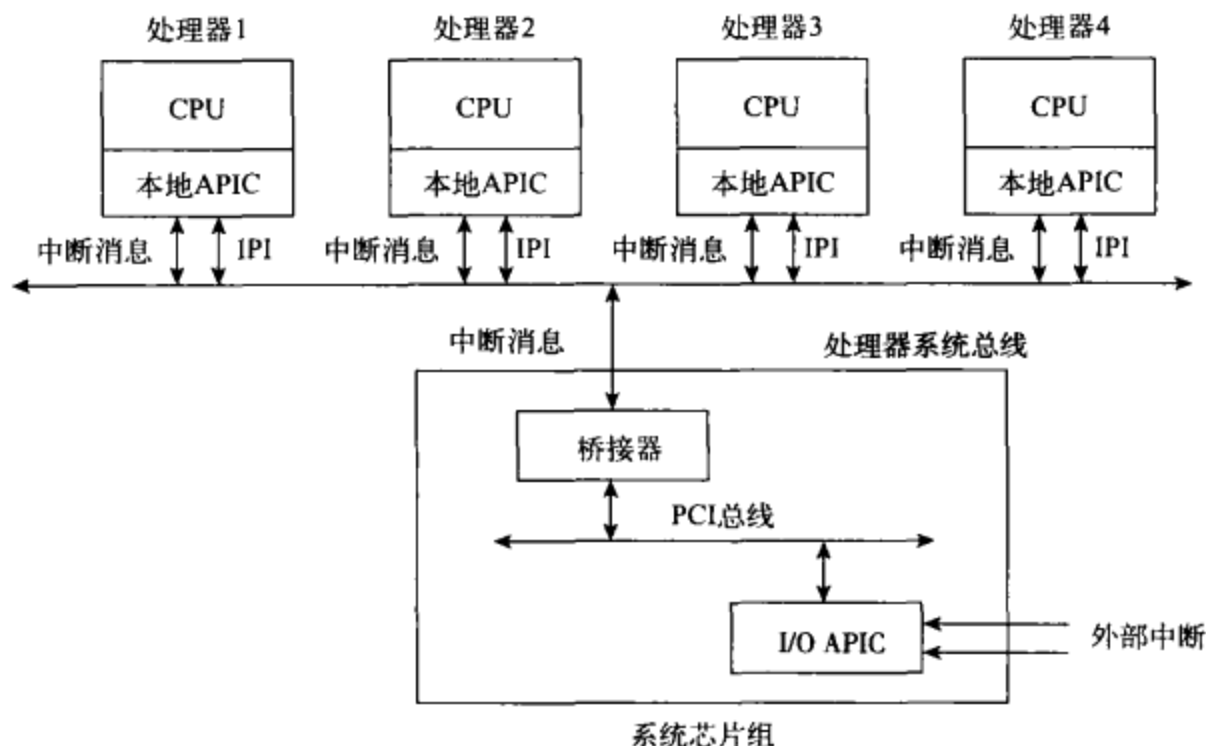


图 20-10 英特尔多处理器 (Xeon) 里的本地 APIC 和 I/O APIC

在目前的建置中,系统的每一个部分都是经由 APIC 总线连接的。“本机 APIC”为系统的一部分,负责传递中断至指定的处理器;举例来说,当一台机器上有三个处理器则它必须相对的要三个本机 APIC。自 1994 年的 Pentium P54c 开始 Intel 已经将本机 APIC 建置在它们的处理器中。实际建置了 Intel 处理器的电脑就已经包含了 APIC 系统的部分。

系统中另一个重要的部分为 I/O APIC。系统中最多可拥有 8 个 I/O APIC。它们会收集来自 I/O 装置的中断信号且在那些装置需要中断时传送信息至本机 APIC。每个 I/O APIC 有一个专有的中断输入(或 IRQ)号码。Intel 过去与目前的 I/O APIC 通常有 24 个输入,其他的可能有多达 64 个。而且有些机器拥有数个 I/O APIC,每一个分别有自己的输入号码,加起来一台机器上会有上百个 IRQ 可供中断使用。

当然，除了处理处理器间及输入输出的中断外，APIC 也负责处理本地中断源发出的中断：如本地连接的 I/O 设备、时序中断、性能监视计数器中断、高温中断、内部错误中断等。

20.6 SMP 缓存一致性

由于对称多处理器结构下，每个处理器都有自己的缓存。这样在一个系统里面存在多个缓存的情况下就有可能出现两个缓存的数据不一致的情况。即两个 CPU 缓存同样的数据，其中一个或两个 CPU 对数据进行了修改而造成两个 CPU 缓存数据的不同。而这有可能造成严重的后果。因此确保 SMP 里面的缓存一致性十分重要。

SMP 必须确保对内存地址的访问是最新的数据。为此，必须使用一些特定的缓存一致性策略或模型。而缓存一致性策略在学术界得到了广泛的研究，并存在许多现成的模型或方法。例如，英特尔公司的 x86 体系结构使用所谓的 MESI 模型来实现缓存的一致性。

MESI 是英语 4 个单词的首字母缩写：Modified, Exclusive, Shared, Invalid。这 4 个字母分别代表缓存的 4 个状态：修改、独享、共享和无效。这 4 个状态的意思，如表 20-1 所示。

表 20-1 MESI 状态及其描述

状 态	描 述
Modified 修改	缓存被写入。该状态将提醒缓存子系统对系统总线进行探测，并在探测到对相应内存单元的访问时将缓存内容写入内存
Exclusive 独享	该状态表明在系统中其他缓存没有缓存该数据拷贝，因此，该缓存内容属于相应 CPU 独享，即可自由进行任何操作
Shared 共享	表示该内存可能存在于多个缓存，但这多个缓存及其对应的内存单元的数据版本一致，即该数据为清洁拷贝（clean copy）
Invalid 无效	复位后的初始状态，表明该缓存的内容无效，需从内存读取

而 MESI 的 4 个状态之间的转换关系，如图 20-11 所示。

MESI 模型下的状态转换在分布式系统里面有着详细解释。读者可查阅分布式操作系统教材或书籍，本书在此不作赘述。

20.7 多处理器、超线程和多核的比较

多处理器、超线程和多核的共同点是均为了提升计算机性能而设计、均可以同时执行多个指令序列。但是区别也是明显的，主要体现在同时执行的两个线程之间共享物理资源的多少。多处理器的共享物理资源最少，每个线程有自己单独的处理器；超线程共享最多，ALU、FPU、MSR、缓存等均为共享物理资源；而多核则介于二者之间，共享处理器，但不共享 ALU、FPU 等。具体来说，我们有：

对于 HT 线程来说，其共享的资源包括 ALU、某些 MSR 和缓存；而其独享的资源有本地 APIC、通用寄存器、L1 缓存、CPUID 等。

第21章 多核环境下的进程同步与调度

引子

公元前146年，经过120年的胶着，罗马终于在第三次交战中战胜了迦太基，成为地中海的新霸主。这座700多年前由台伯河岸边一座小城发展起来的罗马共和国似乎踌躇满志，意气风发，势不可挡地向着世界霸主的权位推进……

但是，随着时间的推移，罗马共和国的前景却日渐暗淡。在繁荣富贵中，传统的纪律和品格逐渐瓦解。曾经帮助罗马成为地中海霸主的2名执政官以及加元老院的管理制度现在已经成为妨碍罗马共和国继续发展的障碍。由于三权分立，谁也不能控制谁。这样，在道德水准江河日下的情况下，内乱频生就无法避免了。

公元前60年，内乱达到了巅峰。罗马一位征战高卢的年轻指挥官，尤里乌斯·凯撒拒绝了元老院让其解散军队的请求，率军队越过卢比孔河，向执政官庞培、克拉苏，向元老院，向罗马共和国公然宣战！并随后击败了庞培和元老院组织的抵抗，占领了罗马。在罗马宏伟的竞技场上，尤里乌斯·凯撒宣告自己为罗马的终身独裁者。至此，存在了800余年的罗马共和国宣告结束，罗马帝国的时代得以开启……

但罗马帝国的开始并不太平。尤里乌斯·凯撒执政仅16年就遭到刺杀，致使罗马帝国又一次陷入内乱。在这次内乱中，尤里乌斯·凯撒的侄子屋大维击败对手马可安东尼和埃及艳后克莉奥佩特拉，成为罗马帝国的新统治者。这位奥古斯都大帝在公元前27年恢复了共和制，并将自己的地位从“皇帝”降低为“罗马首席公民”，从而开创了罗马帝国的辉煌时代。罗马的版图和影响力空前壮大。正如有着“罗马的荷马”之称的诗人维吉尔所说：

哦，罗马人！

请记住，用你的影响力统治各个国家，这应该是你的艺术

用法律成就和平，宽恕乞求之人，用战争制服傲慢之徒……

——摘录《埃涅阿斯记》

繁荣虽得，却不能长久。由于罗马国土过于庞大（如图21-1），284年，罗马皇

帝戴克里先设立了由2名共治皇帝和2名凯撒助手组成的4人执政体系。而这种有着多个执政“核心”的体系很快就显示了巨大的破坏力。执政的4人常各怀鬼胎，导致大规模内乱和东都君士但丁堡的建立。395年，皇帝狄奥多西一世将帝国分送给两个儿子，从此罗马分为东西两个帝国，首都分别为罗马和君士但丁堡。这次分裂加速了罗马的陨落。476年，在盎格鲁、撒克逊、法兰克、伦巴第、匈奴、东西哥特、汪达尔、条顿等凶猛蛮族的多路袭击下，西罗马帝国陷落，罗马时代宣告结束。

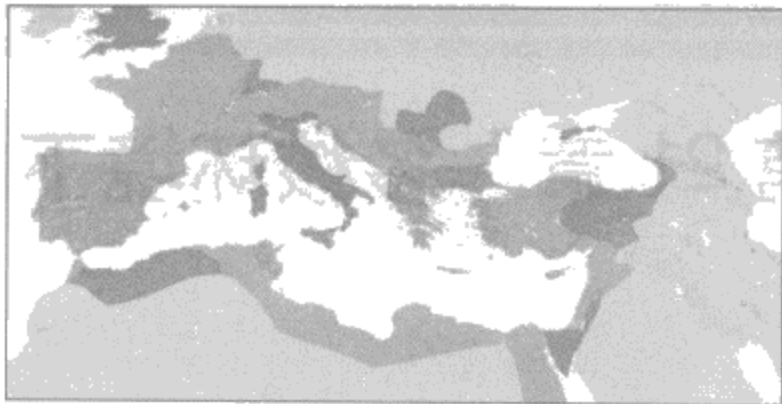


图 21-1 古罗马帝国鼎盛时期的领土范围

而“多核（多个执政核心）”是罗马陨落的一个重要原因。

21.1 多核环境下操作系统的修正

多核的出现对软件的设计产生了巨大的影响。那些原来在单核环境下合理的设计和单核环境下的原语操作将不能适应，而需要修正。除此之外，进程的调度也是一个大大需要修改的地方，因为现在进程不只有一个执行核可以选择了。

内存管理的变化是多核环境带来的另外一个重要变故。此外，能耗优化也是多核环境下必须考虑的问题：如果一个核上没有执行程序，是否需要让这个核停止工作？

本章即对多核环境下的进程同步与调度、内存管理、核能耗优化进行论述。

21.2 多核环境下的进程同步与调度

多核环境带来的最大变化是进程的同步与调度。由于进程运行在不同的CPU或执行核上，其同步就不仅仅是线程的同步，而有可能是执行核或CPU之间的同步。而进程的调度也将涉及到将何进程分配到何CPU或执行核上。由于不同的核在内存的共享方式上有可能不同，其运行有数据共享的进程和没有数据共享的进程的效率就会有很大不同。这就需要调度策略的合理选择与执行来保证系统的整体运行效率。

21.3 多核进程同步

在单核环境下，一个时候只可能有一个程序在执行。而在多核环境下，由于多个执行核或

CPU 的存在, 多个程序可以真正地同时执行。因此, 多核环境下的进程同步与单核环境下将有着很大的不同。

我们先来看一下多核环境下的一个具体的同步问题。假定我们有两个处理器 CPU0 和 CPU1, 而这两个处理器执行的程序均需要对内存地址 %eax 的内容进行加 1 的操作, 如图 21-2 所示。

时间顺序	CPU 0	CPU 1
1.	Read (% eax) to A	Read (% eax) to A
2.	A = A + 1	
3.		(A = A + 1)
4.	Write A to (% eax)	
5.		Write A to (% eax)

图 21-2 CPU0 和 CPU1 同时进行 INC 操作

显然, 如果 CPU0 和 CPU1 对这段程序的执行按照图 21-2 所示错开, 则内存地址 %eax 里面的内容只被加了一次 1。而这与我们预期的结果不符。

要保证上述两端程序执行的正确性, 我们需要在 CPU0 执行其程序时, CPU1 不会执行上述程序。或者 CPU1 执行的时候 CPU0 不会执行。这就需要采取措施保证一段程序的执行是原子操作。不过这里的原子操作与我们前面单核环境下的原子操作有所不同: 它必须保证跨越 CPU 的原子性。即一个 CPU 执行时, 得让另一个 CPU 不执行某段程序。

那么我们如何做出这种保证呢?

正如我们在第 4 章讲过的, 所有软件原语操作均是构建在硬件原子操作的基础上。没有硬件的原子操作, 软件原语操作就是空中楼阁, 无法挺立。在这里也不例外, 所有的多处理器原语操作也需要硬件的支持。下面我们看一下从处理器的架构上面提供了哪些原子操作。

21.4 硬件原子操作

在单核环境下, 我们说过, 硬件提供的原子操作有: 中断的启用与禁止、加载存入指令、测试与设置。而这三种操作除中断启用与禁止不工作外, 其他两种在多核环境下均可使用。

对于加载存入原子操作来说, 下面的操作均是原子操作:

- 读写一个字节。
- 读写一个按 16 位对齐的 16 位的字。
- 读写一个按 32 位对齐的 32 位的双字。

而测试与设置则需要针对共享内存单元进行。

21.5 总线锁

在多核环境下, 还有一种硬件原子操作称为总线锁。总线锁就是将总线锁住, 只有持有该锁的 CPU 才能使用总线。这样, 由于所有 CPU 均需要使用共享总线来访问共享内存, 而总线的锁

住将使得其他 CPU 没有办法执行任何与共享内存有关的指令，从而保护数据的访问是排他的。

除此之外，硬件提供的另外一种同步原语是所谓的交换指令，即 xchg (exchange)。该指令可以以原子操作完成在寄存器和内存单元之间的内容置换。该指令的语义可由下述程序片段表示：

```
int cmpxchg(addr, v1, v2) {
    int ret = 0;
    /* 停止所有内存活动并忽略所有中断 */
    if (*addr == v1) { *addr = v2; ret = 1; }
    /* 重启内存活动和响应中断, 返回 ret; */
}
```

21.6 多核环境下的软件同步原语

在硬件提供的同步原语基础上，我们就可以构建软件同步原语了。由于多核技术相对比较新，如何实现多 CPU 同步尚没有统一标准，这样造成不同的操作系统实现的软件同步原语不尽相同。下面我们看一下 Windows 和 Linux 内核里提供的一些原子的操作。

Linux 内核提供的原子操作包括如下几种：

- 总线锁：置换，比较与置换，原子递增操作。
- 原子算术操作：原子读、设置、加、减、递增、递减、递减与测试。
- 原子位操作：位设置、位清除、位测试与设置、位测试与清除、位测试与改变。

Windows 内核提供的原子操作包括如下几种：

- 互锁操作 (Interlocked Operation)。
- 执行体互锁操作 (Executive Interlocked Operation)。

这里需要注意的是，目前操作系统还没有为多核环境提供锁操作，因为这种操作代价比较大。除了一些特殊的类型，我们还可以对字位操作保证原子性。不过位运算并不是没有代价的，如果能够不共享的话就不要共享，这样可以提高性能。

21.7 旋锁

旋锁 (spin lock) 是几乎所有多核操作系统均会提供的一种 CPU 互斥机制，是操作系统内核用于多处理器互斥的机制，即用户程序不能使用旋锁来进行互斥。旋锁通常用于保护某个全局的数据结构，如 Windows 里面的 DPC (延迟过程调用) 队列。这里的互斥指的是多个处理器或执行核之间的互斥，即两个处理器或核不能 (物理上) 同时访问同一个数据结构；而不是第 4 章讲过的多线程之间的互斥。对于局部数据结构来说，则因为只在一个 CPU 下而不需要使用旋锁。例如，设备驱动程序需要通过旋锁来保证对设备寄存器和其他全局数据结构访问的排他性，即任何时候只能有设备驱动程序的一个部分，从某一个处理器，访问这些寄存器和数据结构。

旋锁通过获取和释放两个操作来保证任何时候只有一个拥有者。旋锁的状态有两种：要么是闲置的，要么被某个 CPU 所拥有。这里再次提醒，旋锁的拥有者是 CPU，而不是线程。因此，如果一个 CPU 获得一个旋锁，那么运行在该 CPU 上的所有的线程都可以访问该旋锁所保

护的寄存器和数据结构。旋锁的使用与 Windows API 里面的 mutex 使用非常类似。

使用旋锁的过程如下：

- 1) 等待旋锁变为闲置。
- 2) 获得旋锁。
- 3) 访问寄存器和全局数据结构。
- 4) 释放旋锁。

例如，Windows 使用旋锁保护对 DPC 队列的访问过程，如图 21-3 所示。

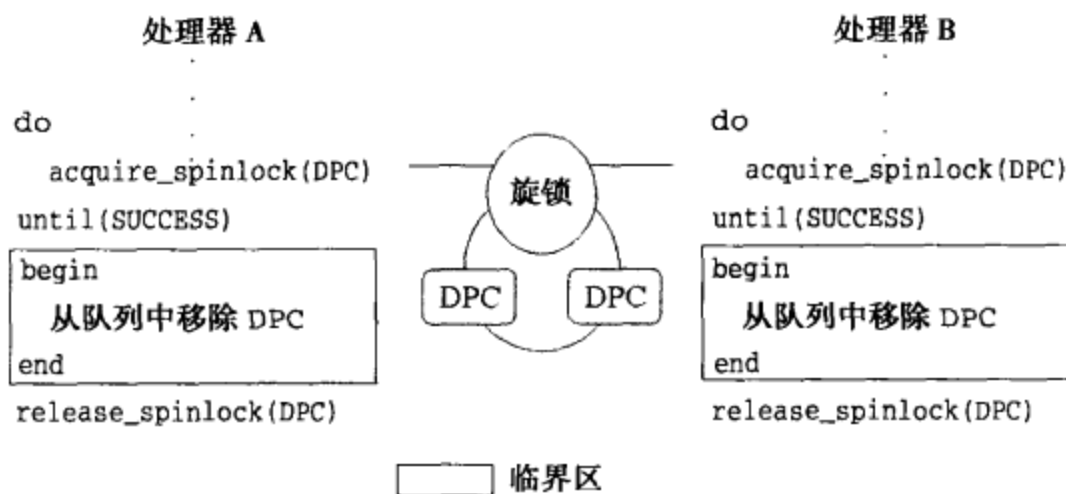


图 21-3 在 Windows 下使用旋锁保护 DPC 队列的过程

在图 21-3 中，两个处理器 A 和 B 均需要访问全局的 DPC 队列（DPC 是延迟过程调用的缩写。主要用于在中断时将那些不需要高优先级执行的代码放进一个队列，等有空时再执行的机制）。因此我们用旋锁来进行处理器间的互斥。对于处理器 A 来说，如果要访问全局数据，就要先获得 Spinlock，直到成功，然后才访问。对于处理器 B 来说情况也一样。

21.7.1 旋锁的实现

旋锁的实现也必须在硬件提供的原子操作上进行。我们前面说过，多处理器环境下的硬件原子操作有加载与存入、测试与设置。这两种方法皆可以用来实现旋锁，而使用测试与设置更为简单。

在使用测试与设置来实现旋锁时，旋锁是一个特定的内存单元。这个特定的内存单元必须位于整个系统的共享内存里面。这是旋锁的物理载体。如果一个处理器要使用旋锁，就必须检查这个特定内存单元的值。如果为 0，则将其设置为 1，表示获得该旋锁。如果为 1，则表示该旋锁被其他处理器所占有，则在该旋锁上进行繁忙等待，即不停地循环，这也是为什么叫旋锁的缘故，如图 21-4 所示。

获得和释放旋锁的代码是用汇编语言写的。如果使用高级语言，有的动作就无法执行，即使能够执行，也很可能速度缓慢。而且体系结构的一些优点也只有汇编语言能利用。为了提高速度并且最大限度地利用底层处理器结构提供的各种锁机制，用来获取和释放旋锁的代码通常用汇编语言写成。

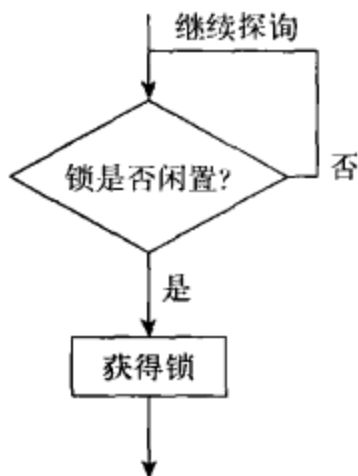


图 21-4 旋锁中的旋转

21.7.2 旋锁的缺点

旋锁有什么缺点呢？很多读者可能马上注意到旋锁的繁忙等待问题。如果这个 CPU 没有获得旋锁，就循环往复，繁忙等待。我们说过繁忙等待不好。那么 CPU 在这儿繁忙等待，是不是一种浪费？本书第 9 部分讨论锁的时候提到过锁的两个缺点：一是浪费资源，二是可能造成优先级倒挂。

虽然我们前面说过，繁忙等待不是什么好事情，但在旋锁的情况下，繁忙等待并不会造成什么不良后果。这是因为，这里繁忙等待的主体是 CPU，而不是线程。

我们先来看一下旋锁里的繁忙等待是否浪费时间。如果没有获得旋锁的 CPU 不进行繁忙等待，那又能干什么呢？CPU 要获取旋锁是因为 CPU 需要使用旋锁才能继续执行，它即使不等在旋锁上，也干不了别的事情。因此，等在上面算不得什么巨大浪费。

这里需要注意的是，竞争旋锁的主体不是线程或进程，会阻碍别的线程运转。这里的竞争主体是 CPU。而对于 CPU 来说，你等待你的，我运转我的，并不会造成系统效率大幅度下降。

另外，在旋锁上的繁忙等待时间通常较短。因此持有旋锁的 CPU 肯定在往前推进（而不像线程持有锁的情况，线程有可能没有推进），会很快释放旋锁。你切换到别的线程再回来，反而慢了。因此，这种等待浪费也不是什么大不了的事情。

锁的另外一个缺点在旋锁上也不会发生。因为旋锁由 CPU 持有，而不是线程持有，因线程具有优先级关系而可能出现的优先级倒挂现象在旋锁上自然也不会出现。

那么旋锁有没有缺点呢？

答案是肯定的。我们说过，人造的东西不可能没有缺点。旋锁也不例外。只不过，旋锁的问题与普通锁的问题不一样。旋锁的问题是总线的竞争。而这是一个严重的问题。

那么总线竞争是怎么一回事呢？

旋锁的物理体现是什么？内存单元。这个内存单元在什么地方？在全局内存里。因为，每个 CPU 在检查旋锁的状态时均需要使用系统总线来访问旋锁所在的共享全局内存单元。如果他要测试并设置这个全局内存单元，就要使用内存总线。那么他不停地发信号到总线上，会造成内存总线的竞争。这是非常浪费，效率非常低下的一种机制。这就是旋锁的缺点。

那有什么办法解决旋锁的总线竞争问题呢？有，解决方案就是队列旋锁。

21.7.3 队列旋锁

队列旋锁的中心思想就是，需要旋锁的 CPU 不要到全局内存去 SPIN，而是到自己的局部内存去 SPIN。这样就可以排除对总线的竞争。我在旋锁上排一个队，就表示哪些 CPU 要这个旋锁，释放的时候就去检查这个队列，交给队列里的第一个 CPU。

那么什么叫“交给对方”？就是把局部变量改成释放状态。你本来是在一个全局单元等着检查，现在你不在全局上等，而在局部单元上等着发生变化。谁来做这个变化呢？不是你自己，而是释放旋锁的 CPU 来改变。一发生改变，你马上就知道了。

那么旋锁的队列在什么地方？那当然在全局的地方。别的 CPU 可以访问局部内存，但是

延迟长。原则上不访问别人的内存，除非迫不得已。这里就是迫不得已。那么这也不会有总线竞争了，因为不会把内存信号发到总线上去。另外这还提供了先进先出的语义，从而在某种程度上达到公平的效果，即使用同一旋锁的 CPU 具有类似的性能。

使用队列旋锁的一个巨大优点是其扩展性。由于每个 CPU 等在自己的本地内存单元上，此种机制几乎可以无限扩展。

21.8 其他同步原语

在多核环境下使用的其他同步原因包括信号量、内核对象等。这些有的在第 7 章已经论述，有些属于商业操作系统的实际实现，本书就不予以赘述了。

21.9 多核环境下的进程调度

对于进程和线程来说，多核环境与单核环境的最大不同是可以有多个线程或进程真正地同时执行，而在单核情景下，这个同时不是真正的物理同时，而是虚拟的逻辑同时。就是这个从逻辑同时到物理同时的变化，使得多核环境下的调度与单核环境下的调度有所不同。

下面我们从调度目标、调度策略、调度算法和调度手段分别对多核环境下的调度进行讲解。在调度目标上，单核环境下调度应该达到的目标，多核环境下也应该达到。此目标包括：

- 具有快速响应时间。
- 保证后台工作的高吞吐率。
- 防止进程饥饿。
- 协调高低优先级进程。

除此之外，多核调度还需要考虑多核之间的负载平衡问题，即每个核的工作量应该比较均衡。

21.9.1 调度策略

对于多个核来说，每个 CPU 有着自己的就绪队列 `runqueue`。该队列里面又可以按照不同的优先级分解为多个子队列，就像单核环境下的情况一样。一个进程可以排在任何一个 CPU 的队列上，但也只能排在一个 CPU 的队列上。图 21-5 描述的就是 Linux 实现里面的一个 CPU 的就绪队列和排在其上按优先级分为不同子队列的情况。图中“活动队列”上排的是可以执行的进程，“过期队列”上排的是不能被执行的进程（受阻进程）。

显然，对于不同优先级的进程来说，其调度算法为优先级高的优先。在同一优

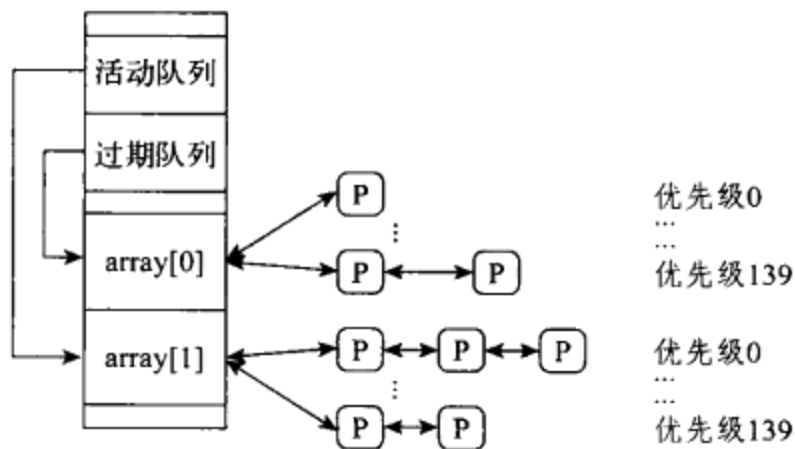


图 21-5 Linux 实现下的 CPU 就绪队列

优先级里面，通常采用时间片轮转。例如，对于图 21-6 里面的 P0 进程来说，其时间片用完后，将被放到其所在队列的末尾，如图 21-6 和图 21-7 所示。

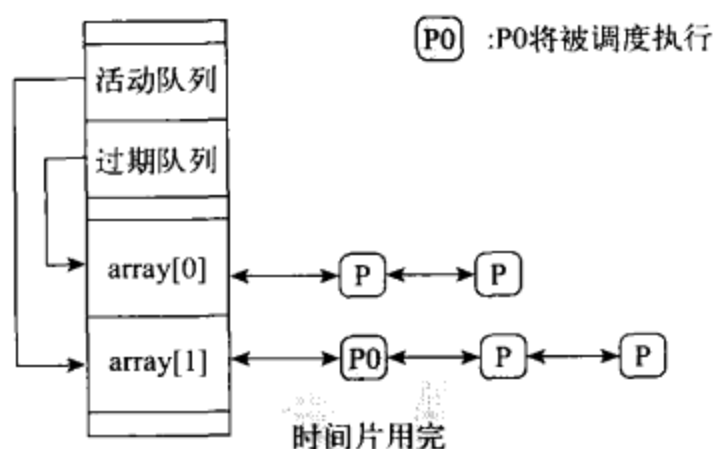


图 21-6 Linux 实现下的时间片轮转调度，P0 的时间片用完

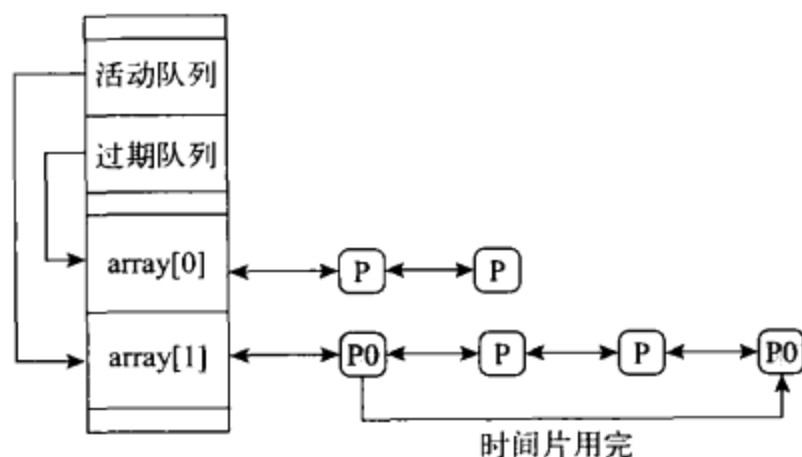


图 21-7 Linux 实现下的时间片轮转调度，P0 被挪动到队列末尾

当然了，与单核情况下类似，一个进程的优先级是可以随着时间的推移而变化的，用以保证进程不会发生饥饿。事实上，对于一个 CPU 的这些队列里面的进程来说，本书在第 8 章阐述过的所有进程调度算法均可用上。这里不再赘述。

如果需要，一个线程也可以从一个 CPU 上移动到另一个 CPU 上。

21.9.2 调度域

我们前面说过，多核环境下需要考虑不同 CPU 之间的负载平衡问题。比如有的 CPU 会很闲，而有些 CPU 会很忙，这就要平衡。而对于 Linux 来说，用于负载平衡的机制是所谓的调度域。调度域指的是一组 CPU。而负载平衡就是在一个调度域里面的 CPU 之间进行平衡，使得它们执行的进程数相同或类似，或者它们的繁忙程度类似。

一个多核系统里面可以有多个调度域。所有的 CPU 均被映射到某个调度域。而调度域是一个层次架构，顶级调度域囊括所有的 CPU，而子调度域则通常仅包括部分的 CPU。例如，一个典型的调度域架构可以如下：

- CPU_domains (HT)
- Core_domains

- Phys_domains
- Node_domains
- Allnodes_domains

每个调度域设有多个标志。这些标志用来表示该调度域的各种属性。主要的标志如下：

```

• #define SD_LOAD_BALANCE 1      /* 对本域进行负载均衡 */
• #define SD_BALANCE_NEWIDLE 2   /* 在变为闲置时进行负载均衡 */
• #define SD_BALANCE_EXEC 4      /* 在调用程序时进行负载均衡 */
• #define SD_BALANCE_FORK 8      /* 在创建子进程时进行负载均衡 */
• #define SD_WAKE_IDLE 16        /* 任务唤醒时进入闲置状态 */
• #define SD_WAKE_AFFINE 32       /* 任务唤醒到醒来的 CPU 上 */
• #define SD_WAKE_BALANCE 64     /* 在唤醒时进行负载均衡 */
• #define SD_SHARE_CPUPOWER 128  /* 域成员共享 CPU 能供 */

```

这些标志都是 Linux 定义的。每个 CPU 有个进程队列,可以通过它计算每个 CPU 的负载。

21.9.3 负载均衡

负载均衡的目标是将进程均匀分配到每个 CPU 的就绪队列里面。一个负载均衡的系统的效能通常会优于一个负载不平衡的系统。对于 Linux 来说,负载均衡在调度域里面进行。负载均衡的方法有主动和被动两种。

主动负载均衡是队列里面进程数多的 CPU 将某些进程推出去,即所谓的 push。被动负载均衡则是队列为空的 CPU 从别的 CPU 队列里面将进程拉出来,即所谓的 pull。

由于一个系统里面有不同的调度域,不同的调度域其繁忙程度有可能不同。因此,在调度域里面进行负载均衡的情况下,也可能需要在不同调度域之间进行平衡。

因此,在负载均衡时,我们需要找出最繁忙的调度域,在每个调度域里面找出最繁忙的队列,然后将任务从一个队列移动到另一个队列,或者另一个调度域。

21.9.4 进程迁移

在判断需要进行负载均衡后,就需要将一个进程从一个处理器队列移动到另外一个处理器队列。而这个移动包括了整个上下文的移动,如页表、TLB、缓存等。

21.9.5 钉子进程

有时候,因为一个进程的特殊性,我们需要让它在特定的 CPU 上执行。这在非对称多处理器的多核环境下非常普遍。在这种情况下,由于不同 CPU 的功能并不相同,而一个进程可能需要某个特定 CPU 的功能才能执行。即使这个 CPU 非常繁忙,我们也不愿意将某个进程移动到另一个 CPU 上。这个时候我们就将一个进程钉在一个 CPU 上,即所谓的 pined。一个被钉住的进程是不能移动的。

除了因功能不同而钉住一个进程外,还可以因为缓存而钉住一个进程。如果一个进程的许多信息已经缓存在一个 CPU 的缓存里面,我们可能也不太愿意迁移该进程到别的 CPU 上。

21.9.6 关联线程的调度

如果一个应用被分解为多个线程,由于多个线程需要共享许多资源,这个时候需要将这些线程尽量分配到同一个处理器核上执行,以提升缓存命中率。

最后需要指出的是,对于多核处理器系统的调度,目前还没有公认的或明确的标准。因此,各个相关的实体在此方面都是各说各话。因此,本书论述的调度方法只能做一种参考。

21.10 多核环境下的能耗管理

能耗已经成为信息领域面临的一个重大问题,近年来,如何降低信息系统的能耗已经成为一个热门的研究课题。对于多核计算机来说,降低能耗比在单核时更加重要。

通常情况下,CPU 运行在正常状态,其主频时钟频率运行在最高值,此时的能耗也处于最高状态。但如果一个 CPU 没有任务可执行,我们可以令其执行一条所谓的终结(halt)指令,使其进入到 C 状态(一个耗能很低的状态);如果一个 CPU 的工作量很少,我们可以降低其主频频率,使其运行在较低的速度上,从而降低能耗。这个低频率状态就是所谓的 P 状态。

不过,在进行状态改变的时候,需要考虑到许多因素。这些因素包括:逻辑 CPU 之间的依赖性、不同物理 CPU 之间的相关性。例如,同属于一个物理 CPU 的逻辑 CPU 是否可以独立地改变状态?一个 CPU 状态地改变是否会影响其他物理 CPU 的运行?这些相关性因多核结构的不同而不同。我们下面看一下超线程和多核结构的情况。

21.10.1 超线程结构

对于超线程结构来说,C 状态是独立的。即一个逻辑 CPU 可以变为 C 状态,而不影响其他逻辑 CPU 的运行。直白地说,我们可以彻底关闭一个逻辑 CPU,而将其所用资源全部移交给其他逻辑 CPU。如果只有两个逻辑 CPU,则关闭一个逻辑 CPU 的结果相当于暂停多核环境。

P 状态是不独立的。因为一个物理 CPU 里面的逻辑 CPU 共享同一个时钟,它们的 P 状态控制寄存器也是共享的。因此,一旦一个逻辑 CPU 进入 P 状态,则另一个逻辑 CPU 也将进入 P 状态。事实上,所有共享 P 状态控制寄存器的逻辑 CPU 皆进入 P 状态。

21.10.2 多核结构

对于多核结构来说,C 状态是独立的。即一个核可以独立的进入到 C 状态而不影响另一个核的运转。如果两个核同时进入 C 状态,则两核之间的共享部分也可以进入 C 状态。如果双核里面只有一个核进入 C 状态,则共享部分不能进入 C 状态。

多核之间有着独立的 P 状态控制寄存器,但是每个核之间的 P 状态是相互依赖的,它们之间的依赖关系由软件协调。

21.10.3 调度上的考虑

关闭一部分执行核可以降低系统的能耗。但关闭执行核或降低其执行频率有可能对与其有关联的其他执行核产生影响。因此,为了既可以节能,又不对其他执行核产生影响,我们可以从调度上予以考虑。例如,将任务分解到一个包装里面的逻辑 CPU 上,使得这一个物理 CPU 上的所有逻辑 CPU 都处于忙碌状态,而其他物理 CPU 保持空闲,从而可以关闭或降低执行等级。只有无法将任务都置于一个包装的逻辑 CPU 时,才分解到其他物理 CPU 上。

还有就是多核的电源优化。CPU 没有执行任务的时候,会执行一个指令,使 CPU 进入不工作的状态。但这个状态能够保留寄存器里面的数据。例如,在 Linux 操作系统下,每个核可以有 C1、C2、C3 三种状态。C0 为正常状态,C1、C2 为电源优化状态。我们可以通过将不同的核设置为不同的状态而达到电源优化的目的。比如一个可以在 C3 状态,一个可以在 C0 状态。即使是在 C0 工作状态,也可以通过调节执行核的频率和电压来省电。

21.11 讨论:多核系统的性能

多核真的能提高计算机的性能吗?答案可不一定。

就像衡阳保卫战的情景一样。虽然中国军队在衡阳外围有 6 个军的兵力,但由于每个军的指挥独立,相互之间不能协调,导致救援失败,致使衡阳最终沦陷。

对于多核来说,每个核可以独立的执行任务。而这些核之间是否可以协调和在何种程度上进行协调就决定了多核系统的性能提升到底有多少。

随着多核技术的普及,越来越多的人拥有了自己的多核计算机。可是很多人并没有觉得多核计算机在速度和性能上有着任何显著的提高。有的人甚至根本就没有感觉到任何效率上的变化,有的人甚至觉得效率反而下降了。

这些人的感觉是对的吗?

我们当然知道,感觉是靠不住的东西。因为没有觉得效率的提升并不等于效率真的没有提升。也许是我们的期望值太高,在期望与现实的反差之下对效率的提高视而不见;也许是在多核计算机上运行的软件更为复杂,从而屏蔽了效率提高的现象;也许我们十多年前就开始使用多核计算机了(例如,Sun 的 SPARC),因此,对现在英特尔、AMD 等推出的多核技术不屑一顾;也许我们对多核还有敌意,故意诋毁多核在此方面的作用。

但是我们也知道,感觉并不是无中生有的,很多人说没有体会到效率的提高也不是空穴来风。事实上,多核真的不一定能提升你的软件执行效率。而对于某些特殊应用,如嵌入式应用、高性能计算,多核非但不提高性能,反而将降低效率。例如,对于嵌入式来说,一家电信设备制造商将其应用从一个多处理器系统迁移到一个多核系统。而这个迁移所带来的区别只不过是独立内存(多处理器)到共享内存(多核),造成了巨大的性能下降。而且这种下降不以应用本身的并发程度为转移(参看 IEEE Computer 2008 年 11 月刊:Measuring Multicore Performance, p99-102)。

2008 年,位于新墨西哥州的美国圣地亚国家实验室(Sandia National Laboratories)对多核技

术进行了大规模的仿真试验,他们对 8 核、16 核、和 32 核计算机系统进行了试验,试验结果令人失望。结果表明:随着核的数量增加,多核计算机的性能很快就不再提升,并在到达一定数量后,性能呈下降趋势。尤其是对于需要进行大量数据处理的应用如 informatics applications (如海量数据筛选、安全控制系统)来说,这种性能下降十分显著。对于这些应用来说,核的增加并没有带来性能的增加。根据圣地亚国家实验室的数据,在核数达到 8 以后,核数的增加丝毫也不会带来效率的提升(见图 21-8)。

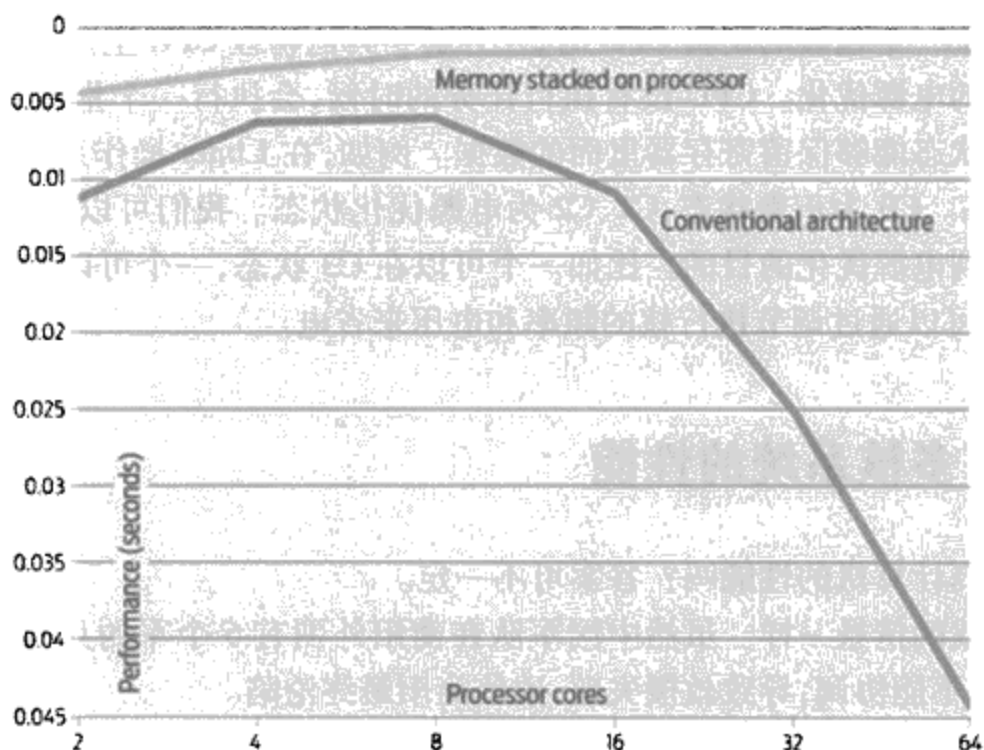


图 21-8 多核与性能关系(来源:美国圣地亚国家实验室)

当然,多核技术并不是一点也不会带来性能的提升。如果这样的话,多核技术也不会发展到如今路人皆知的地步。事实上,多核技术对于那些可以自然分解为大量并发任务的应用来说如庖丁解牛,游刃有余。

由此可见,多核技术到底是否带来益处完全取决于客户的应用是什么。如果是解微分方程,请使用多核计算机,如果是进行海量信息筛选和数据挖掘,则最好不要使用多核计算机。

即使是可以利用多核技术的应用,其利用程度也有赖于程序的编写方式。只有在程序编写时就充分考虑到多核的结构特点的软件才能最大限度的从多核技术获得益处。而充分利用多核结构编写并发度高的程序却不是很多人能够胜任的一个任务。当然,多核厂家或编译器的制作者可以在编译器这一层上动动手脚,将普通程序员编写的非并发程序尽可能编译为某种程度的并发代码,从而对多核技术加以利用。

当然了,即使我们写不出并发程序,也并不意味着多核技术就没有好处。至少,我们在不同的核上面跑不同的应用。当然,这些不同应用之间的联系最好别太多。

造成多核技术这个问题的核心是所谓的内存墙。虽然 CPU 的速度不断提升,但其访问数据的速度却没有同步提升。虽然核数不断增加,但从这些核到计算机其余部件的链接并没有同步增加,这样,保持所有 CPU 获得充足的数据就是一个大问题。

对于多核技术来说,现在的问题是,在信息爆炸的年代,数据挖掘和数据分析的应用越来越

占据重要地位,如果将多核技术引导到能够为这些应用提高有意义的性能提升是一个摆在所有关心多核技术的人的面前一个紧迫课题。多核技术到底能发展到什么程度,就让我们拭目以待吧。

思考题

1. 多核调度时与单核调度最不一样的地方是什么?
2. 简要阐述旋锁的工作原理。
3. 旋锁的持有者是谁? 进程、线程还是其他?
4. CPU 在旋锁上的繁忙等待是否造成优先级倒挂? 为什么?
5. CPU 在竞争旋锁时会通过忙等来等待旋锁的释放。有同学建议,CPU 在发现旋锁繁忙时不要忙等,而是睡觉去。而持有旋锁的 CPU 在释放旋锁时给需要使用旋锁的 CPU 发消息或中断来叫醒 CPU。请问你觉得这个办法如何?
6. 多核系统带来的操作系统变化主要是什么?
7. 将程序从单核环境移植到多核环境下是否一定会缩短运行时间? 为什么?
8. 什么是进程迁移?
9. 多核环境下的能耗管理有哪些措施? 你还能想出别的办法吗?
10. 在支持超线程的多核结构下,一个进程的两个线程是调度一个物理执行核上的两个逻辑执行核好,还是分别调度到两个不同的物理执行核好? 对于两个不同的进程呢?

PART SEVEN

第七篇 操作系统设计 原理篇

前面六大篇对操作系统的基本概念和各种机制进行了详细论述。虽然这些论述是分篇分章进行的,但它们之间并不是互相割裂的,而是有着内部的逻辑联系的。联系它们的纽带有多条。首先是操作系统作为计算机的管理者,需要对计算机的各个组成部分进行管理,这就导致了 CPU 管理、内存管理、磁盘管理、输入输出设备管理等操作系统功能的出现和相互关联。其次,操作系统作为魔术师,需要对计算机的各种硬件进行抽象和装扮,以使其显得更大、更快、更好、更容易使用。而这些抽象就形成了进程线程、虚拟内存、文件系统、各种 I/O 模式等操作系统构造的出现。而这些构造之间也因操作系统魔术师的角色而互相联结起来。

但除了管理和魔幻这两条共同的纽带外,操作系统各个部分还有一个联结纽带:其设计上所遵循的哲学原理。虽然 CPU、内存、磁盘、输入输出等设备听上去相差甚大,但对它们进行管理和抽象时采用的策略却有很多相通点。本篇即对这些相通点或哲学原理进行讨论。

本篇从高屋建瓴的角度对操作系统设计的十条哲学原理进行阐述。显然,操作系统的设计原理有很多,本篇选取的只是这诸多原理里面非常重要的十条。第 22 章将从操作系统和人类社会两个层面对这十条原理进行论述与比较,以使读者更加清楚地明白操作系统就是人类社会在计算机里面的反映,明白了人类社会的运转就明白了操作系统的运转。读完本篇后,读者自己可自行去发现挖掘操作系统的其他设计原则和原理。

本篇最为重要的核心是不同的生活哲学将导致不同的操作系统设计与构造。只要真正理解了人生哲学,就真正明白了操作系统。



操作系统的不同设计从根本上说是不同哲学的对决

第 22 章 操作系统设计之原理

引子

设计师:你好,Neo。

Neo:你是谁?

设计师:我是设计师,我创造了 Matrix。我一直在等你。我知道你有很多问题要问,虽然这个过程改变了你的神志,但你依然是不折不扣的人。因此,我的一些回答你能明白,有些你将不能明白。相应地,你的第一个问题也许是最适当的一个问题,但你没有意识到的是它也是最无关的。

Neo:为什么我会在这里?

设计师:你的生命是 Matrix 设计中固有失衡等式的总和。你是一个异常情况的终极体现。尽管我竭尽全力避免它,仍不能将其消除,这并不让人意外,因此,它并不是无法控制的。而这就是你为什么会在哪里。

Neo:你还没有回答我的问题。

设计师:你说得对。有意思,这个问题你问得比其他的要快。

Neo:其他的? 什么其他的? 有多少个? 回答我!

设计师:Matrix 比你想像的要老得多。我喜欢用异常的出现次数来计算 Matrix 的年龄。以这种方式计算,这已经是第六个版本了。

Neo:那只有两种可能的解释:要么没人告诉过我,要么他们也不知道。

设计师:正确。就像你推测的,这个异常是系统范围的,即使最简单的等式也受其影响。

Neo:选择。问题的关键是选择。

设计师:我设计的第一个 Matrix 非常完美,像一件完美无瑕的艺术品。其成功只有它后来史诗般的失效可以比美。它失败的必然性现在来看非常清楚,就是由每个人的固有缺陷所导致。因此,我根据人类的历史重新设计了 Matrix,以更准确地反映人类本性中多变的怪诞特质。但我再次失败了。我最后终于明白我得不到正确答案是因为它不需要太多的心智,或者说它需要的是一个无需完美的心智。正因如此,问题的解答被

另一个程序偶然发现,而这个程序原本是为研究人类心理而设计的。如果说我是 Matrix 之父,则该程序无疑是 Matrix 之母。

Neo:先见之明。

设计师:嗯。正如我所说的,她偶然发现了一个解决方案。只要你给他们选择的自由,这个方案被将近 99.9% 的试验体接受,即使他们只在潜意识上意识到这个选择。虽然这个方案可行,但却在根本上存在缺陷。因此,它产生了自相矛盾的系统范围的异常。如果不加以抑制就有可能会威胁到系统本身。对于那些拒绝该方案的试验体,尽管只是少数,如果不加以抑制就会不断增加灾难发生的可能性。

Neo:你指的是锡安。

设计师:你在这里是因为锡安就快要被摧毁。居住在里面的人全都会被消灭,那里所有的一切都会被抹去。

Neo:胡说。

设计师:否认是所有人类反应中最容易预测的一种。但请相信,这将是我们的第六次摧毁锡安,并且我们干起来越来越得心应手。救世主现在的作用就是要返回源极,将你所携带的编码重新植入主程序。然后从 Matrix 中选出 16 个女性,7 个男性共 23 个人类个体来重建锡安。不按照这个过程进行将导致灾难性的系统崩溃,而这将杀死连接在 Matrix 上的所有人。加上锡安的毁灭,整个人类将不复存在。

Neo:你不会让这样的事情发生,你不能的。你们需要人类才能生存。

设计师:我们已经作好了接受多种存活水平的准备。但与此相关的问题是你是否已经准备好为这个世界所有人人类的灭亡承担责任?

设计师:你的反应很有趣。你的五个前辈都基于一个相同推测:一个用来建立你与你们物种深刻联系的条件判断。而这个判断将使救世主正常运行。其他救世主们对此的体验非常抽象,而你的经历却是相当具体:你正经历着爱。

Neo:崔妮蒂!

设计师:中肯地说,她进入 Matrix,牺牲她自己是为了救你。

Neo:不!

设计师:终于到了揭示真相的时刻。当系统的根本缺陷最终显现的时候,所体现的异常既是开始又是终结。这里有两道门,你右边的门通往源极和锡安的拯救,左边的门回到 Matrix,回到 trinity,和你们种类的绝灭。就像你充分表明的,问题的关键是选择。但我们已经知道你如何选择,不是吗?我已经能够看到你身上的连锁反应,你体内的生化物质表明情感的发作已经开始,而这种情感像是专门为了压制逻辑和理智而设计的。你的情感使你看不到这个简单而明显的事实——她就快要死了,而你却无可奈何。

(Neo 走向他左边的门)

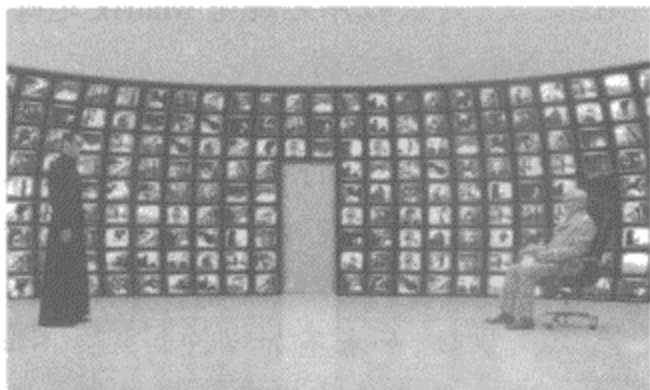
设计师:哼。希望,人类错觉的精髓。它既是你们最强大的力量又是你们最大的弱点。

Neo:如果我是你,我希望我们不会再见。

设计师:我们不会再见面的……

——摘自《黑客帝国 2——重装上阵》(见图 22-1)中设计师与 Neo 的对话

操作系统的设计就是将方方面面的技术和设计有机合并起来,构建一个掌控整个计算机的巨无霸软件系统。当然这种整合并不是一件容易的事情。各种各样的技术细节并不一定互相兼容或相得益彰。它们有时候甚至是矛盾的,或者相消相损的。如果将进程管理、内存管理、I/O 管理、文件管理和安全管理有条不紊地整合到一起,需要花费巨大的心血。



整合所遵循的原则就是操作系统的哲学原理。而这个原理不是别的,就是人类自身特质所推动的人类生活哲学。就像 Matrix 设计师所设计的 Matrix 图 22-1 《黑客帝国 2》中 Neo 与设计师的对话因符合人类自身的怪诞特质而让多数人感到满意一样,只要我们也遵循人类的特质,从人类的生活哲学着眼进行设计,就能够设计出令多数人满意的操作系统。

22.1 操作系统设计的追求

与 Matrix 设计师的追求一样,操作系统的设计追求与人类自身的追求相同,无外乎有如下几个目标:

- 保证操作系统本身运行正确。
- 提供尽可能多的功能。
- 尽量提高系统的效率。
- 在追求效率的基础上尽量顾及到公平。

上面第 3 条的效率有两层意思:一是这个实现的系统本身具有很高的管理和运行效率;另外一层意思是实现过程本身成本很低。

仔细分析可以看出,上述四个追求与人类社会自身的追求完全合拍。人类社会当然要保证社会本身运转秩序正常(社会秩序良好),在此基础上提供尽可能多的功能(社会功能),这个社会必须以效率为导向(例如“让一部分人先富起来”),在追求效率的基础上尽量顾及公平(例如“创建和谐社会”)。由此可见,在操作系统设计时可以将人类社会发展中得出的原理加以应用,从而导出操作系统设计所要遵循的哲学原理。

显然,人类的生活哲学很多,应用到操作系统里面的原则自然也很多。本书不打算将所有设计哲学原理均予以讨论,仅挑选十条作者认为最重要的设计原则进行论述。其他的原理读者可自行体会。

22.2 操作系统设计的第 1 条哲学原理:层次架构

纵观世界的任何一个国家或部落,我们都会发现某种层次结构:一个国家元首或部落首领之下的层层管理机制。每一层的功能各不相同,且下面一层只对上面一层负责。上面的层通常也

只对其直属下层进行直接控制。

操作系统也不例外,它的功能也是分为多个模块,并按层次分解。下面一层向上面一层提供功能,而上面一层也只能对直接下属进行控制,如图 22-2 所示。

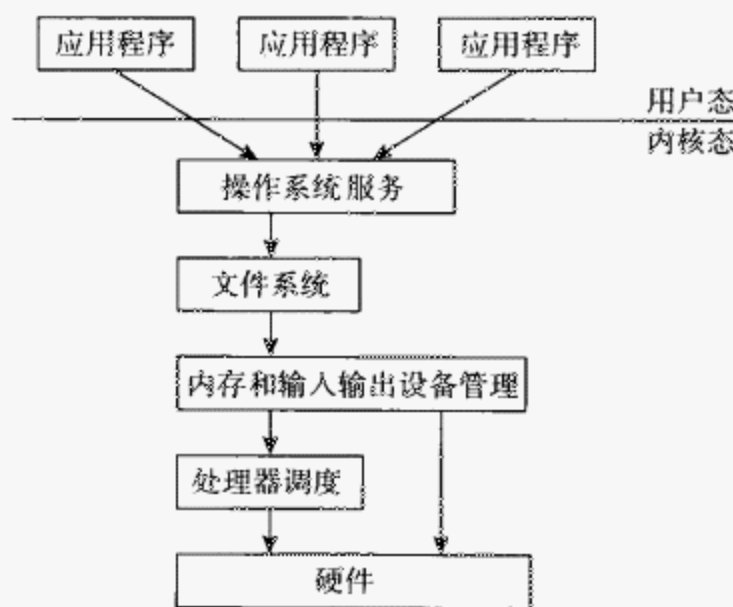


图 22-2 操作系统的层次架构

采用层次结构不仅使得操作系统的构造过程容易,也因为符合人类的习惯而更加易于理解。这样将使操作系统结构清晰,从而节省我们开发操作系统的成本。

22.3 操作系统设计的第 2 条哲学原理:没有对错

人类社会制度发生过多次变化。即使在现在,世界上也存在多种社会制度或形态。但不管社会制度和意识形态如何,我们都可以和平共处。可见,社会形态本身并无对错之分,只有好坏之分。也许某个社会制度给人们提供了更多福利,某个制度具有更高的稳定性,某个制度给人以更多的自由。人类历史上存在过的社会制度都有其存在的合理性,它在某个时代的出现是符合当时的人类思想境界和经济现实的。而人类社会制度的变迁实际上是对效率和公平的追求而导致的。但问题是效率和公平并无客观标准。对于某些人来说的效率,对另外一群人可能是不公平的。

放在操作系统里,这条原则同样有效。操作系统本身并无对错之分,只有好坏之分。就像我们不能说 Windows 是对的,UNIX 是错的。我们只能说,Windows 更容易使用,而 UNIX 不太好使用而已。因此,在设计操作系统时,只要达到功能、效率、公平、正确的平衡即可,而不用担心系统是错的。例如,操作系统进程调度策略有很多,而每种调度策略有其适用的场景。我们不能说,时间片轮转是对的,而优先级调度是错误的。只能说时间片轮转更接近公平,而优先级调度更接近人类社会的等级制度。



图 22-3 操作系统设计就像跳探戈,一种没有对错的舞蹈

没有对错的原理在数据结构里面体现得更为充分。例如,平衡搜索树的实现就存在 AVL 树

(高度平衡树)、Splay 树(伸展树)、红黑树、费波拉齐亚树和 B 树。这些树中每一种树有其使用的场景,但却没有对错之分。在使用红黑树的场景下当然可以使用伸展树或费波拉齐亚树,不会造成正确性的改变,只不过效率有可能不一样。

这种没有对错的原理在风靡一时的探戈里也有充分的体现(见图 22-3)。探戈舞作为所有舞蹈里面最自由的舞,它没有对错之分。无论你做什么动作,在探戈里面都可以,而别的一些舞蹈就不可以随便做动作。因此,从某种程度上说,操作系统的设计就跟跳探戈一样,你觉得好,你就做。

22.4 操作系统设计的第 3 条哲学原理:懒人哲学

有人说,这个世界是由懒惰的人推动的。因为懒惰,我们发明洗衣机来替我们洗衣服;因为懒,我们发明汽车来当行步工具。当然有人不这么认为,他们会说是为了效率。但不管怎么说,很多人都会有一种习惯,就是一件事情不到迫不得已时是不会去做的。这就是懒人哲学。而这一点在操作系统设计时被淋漓尽致地体现出来。我们来看一下 UNIX 操作系统里面的 fork 系统调用的实现。

本书第 3 章已经论述过,fork 是一个类 UNIX 操作系统里面创建子进程的系统调用。在 fork 刚刚出现的时候,fork 创建的是一个和父进程一模一样的子进程,它们有着相同的地址空间和资源。而 fork 的实现就是将父进程地址空间拷贝到子进程地址空间里。这样,在 fork 后,系统里面将出现两个一模一样的进程,执行一样的功能,只不过其进程 ID 不同,执行的序列顺序不同。

一切似乎顺理成章,但问题是,应用程序在 fork 时真的是为了创建两个执行同样任务的进程吗?这种可能性当然存在。但是大部分时候用户 fork 一个进程是为了执行一个不同的程序,如图 22-4 所示。

```

1. while (TRUE) {                                /* 循环往复,以致无穷 */
2.     type_prompt();                             /* 显示提示符 */
3.     read_command (command, parameters)         /* 从终端读取命令输入 */
4.     if (fork() != 0) {                         /* 创建子进程 */
5.         waitpid( -1, &status, 0);             /* 等待子进程返回 */
6.     }
7.     else {                                     /* 子进程代码部分 */
8.         execve (command, parameters, 0);      /* 加载新程序覆盖子进程 */
9.     }
10. }
```

图 22-4 fork 系统调用的使用

在图 22-4 中,fork 出来的子进程 (else 语句) 立即被一个新的程序所覆盖 (第 8 行)。这样,fork 出来的子进程执行的是不同于父进程的程序,这是多数人所希望的。

不过,这样就带来了一个问题:既然子进程被创建后立即就被新的程序覆盖,那么将父进程地址空间的内容拷贝到子进程地址空间显然就是一个浪费。而造成这种浪费是因为我们做事情太积极。还没有搞清出子进程的用途时就将父进程拷贝到子进程。

为了消除这种浪费,工程师们对 fork 的语义进行了修改:在 fork 时只创建一个空的子进

程，而不进行父子进程地址空间的复制。这样，如果将来要运行新的程序，我们没有做任何无用功。而万一用户创建子进程的目的在于运行和父进程同样的程序，则到需要的时候再复制不迟。实际上，fork 的修改使得父子进程共享一个地址空间。只要父子进程均不对地址空间内容进行修改，这种共享就是没有任何问题的。而一旦某个进程（父或子）需要对地址空间进行修改（写操作），这个时候我们将需要修改的值进行复制，使得父子进程使用不同的复制，从而使得写操作可以正常进行。这种不到万不得已不复制在计算机术语里面称为懒惰或延迟的复制（Lazy Copy）。

懒人哲学（见图 22-5）的合理性在于提前将事情做掉也许是一种浪费。因为没有人能够预见到未来会发生什么事情，而情况的变化有可能造成前面所做事情毫无意义。例如如果某个领导的意见经常变化，则下属就可能不必将领导的命令贯彻执行，而是拖延，也许过一段时间，领导改变观点，造成前面的命令无效。这样懒惰执行的结果是节省了能量。又例如，有人将人生规划的完完美美，但也许突然地球爆炸，一切全完，所有提前做的事情都成泡影。



图 22-5 能懒则懒既是生物界的原则，也是操作系统的一条设计原理

操作系统里面的懒人哲学之所以合理也是因为同样的原因。我们并不知道后面会发生什么事情，也许你急于完成的事情过一会儿就变得不值一提了。

22.5 操作系统设计的第 4 条哲学原理：让困于人

中国有句古话：个人自扫门前雪，休管他人瓦上霜。或者说将方便留给自己，困难让给别人（也有人认为是：将困难留给自己，把方便让给别人）。但把方便留给自己是人的本性。虽然我们鼓励人们大公无私，或者为他人着想，但这永远是一个理想，达到这个境界的人即使有，也是凤毛麟角。而大家司空见惯的是所谓的“踢皮球”。这个人踢过来，那个人踢过去，就是没有人愿意费力气帮助需要得到帮助的人。

而作为人所设计的操作系统，人的这一本性自然体现无疑。一个简单的例子就是文件系统一致性的保证。操作系统会使用各种原语操作保证文件系统的一致性，但这仅限于文件系统的元数据，而不是平面数据（用户数据）。即操作系统只对目录夹的操作进行原语保护，而对用户数据的操作通常不会采用此种保护措施。对于操作系统来说，它需要保证自己的正确性，而文件夹对于操作系统的文件系统的正常运转至关重要，因此，文件夹必须保持不能出现问题。而用户文件的一致与否则不影响操作系统本身的运行。虽然用户文件毁坏可能激怒用户，但这不是操作系统有义务管的事情。而这种哲学就是让困于人。

当然了，如果操作系统有能力也有空闲，适当地进行用户数据的保护也是可以的。但请记住，这是份外活，而不是份内事情。

另外一个例子，是对死锁的处理。我们讲过，死锁的应对有四种模式。虽然现代操作系统采取了部分措施来降低死锁的概率，但由于死锁的动态避免代价巨大，现代商业操作系统均不

支持动态避免，从而造成死锁的不可避免。而这种不愿花费力气进行死锁避免，却把死锁可能留给用户的做法就是典型的让困于人，即让用户来承担困难。

如果我们放眼到操作系统之外，这一点体现的更为清楚。例如，在计算机硬件引入了流水线设计后，出现了一个所谓的分支延迟和加载延迟的概念。这是因为一条分支指令发射到流水线后，需要至少两个时钟周期才能判断是否需要分支。而到第2个时钟周期时，后面的一条指令将已经进入流水线。如果这个时候发现需要分支，后面的指令有可能就是不应该执行的指令。这时就需要将后面一条指令从流水线上清除出去。而清除流水线的代价颇高。

这样，分支指令后面的一条指令无论前面是否分支都会被发射到流水线上。这条指令所占的位置就称为分支延迟位（Branch Delay Slot）。而为了避免流水线清除操作所带来的代价，最好的办法就是在分支延迟位上放一条无论是否分支均需要执行的指令。这样无论分支与否，流水线都不需要进行清除操作。

显然，这是一个不错的想法。但问题是，谁有能力判断一条指令的执行是否与前面的分支条件有关呢？用户？编译器？操作系统？答案是显然的。只有编译器有能力作出这种判断。因为只有编译器看过了这个程序，事实上，它看了两遍。自然有条件选出一条不依赖分支条件的指令放置在分支延迟位上。

这样看的话，分支延迟位的问题解决了。但现实是，这个问题没有解决。原因就是编译器不愿意承担这个选择分支延迟位指令的差事。因为这个实在费力，弄不好还会造成程序错误。编译器为什么要承担这个额外的义务呢？既然研究计算机体系结构的人提出来流水线，凭什么让编译器来收拾其弄出来的烂摊子呢？编译器的设计人员当然不会答应。

这样，这个问题最后只好由研究体系结构的人自己解决：或清除流水线或插入空白操作。

22.6 操作系统设计的第5条哲学原理：留有余地

对于大部分人来说，在做任何事情的时候都会留有一点余地。俗话说，不可将事情做绝。虽然兵法里面有所谓的置之死地而后生的说法，但这毕竟是在特殊情况下的一种战术，而不是绝大多数人信奉的教条。对于芸芸众生来说，还是留有回旋余地比较好。

而留有余地这一点在操作系统的设计中也得到了充分体现。看文件系统目录夹的设计：一个目录夹记录里面通常都有一部分所谓的保留空间。例如，DOS目录夹记录里面就有10个字节的保留空间，如图22-6。

8	3	1	10	2	2	2	4
文件名	扩展名		保留空间	时间	日期		尺寸

图 22-6 DOS 目录夹记录里面的保留空间

之所以留下这10个字节的保留字是因为我们不能肯定我们的设计是完美的。这样，如果后来人对此系统进行改善，他们将有余地回旋。而事实上，这10个保留字在Win98文件系统里面就得到了利用。如果没有这些保留字，我们将不得不设计完全新的系统而导致无法兼容。

22.7 操作系统设计的第6条哲学原理：子虚乌有——海市蜃楼之美

操作系统的目的是服务上层的应用程序和用户。而这些上层应用的要求经常是五花八门，吹毛求疵。这些要求和硬件直接能够提供的服务相差十万八千里。架起这十万八千里长的桥梁的不是别的东西，是操作系统。而为了架起这座桥梁，操作系统是用户要什么就提供什么。而提供的这种东西在用户看来虽然实实在在，但实际上都是子虚乌有的东西（见图22-7）。

而这正是操作系统的最大角色：魔术师。就像 Matrix 电影里面，机器人设计了一个巨大的魔幻：一个让人感觉十分真实的物质世界。虽然生活在 Matrix 里面的人都感觉这个世界是真真切切的，但这个物质世界实际上根本不存在，是子虚乌有的。

例如，在操作系统里面，用户看到的内存是一个非常简单、具有线性美的一维数组。这个数组的空间无限大（实际上是和磁盘一样大），速度无限快（实际上是和缓存一样快）。但实际上我们的物理内存只不过 512M 或者 1G（也许更大一点，但终归不如磁盘大），速度也只有缓存的十分之一。因此，用户看到的无限大、无限快的内存空间是根本不存在的。

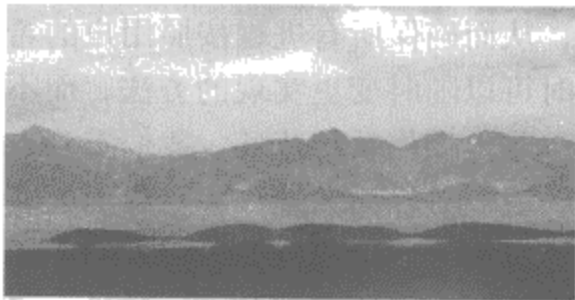


图 22-7 操作系统所提供的各种抽象有如现实生活中的海市蜃楼

22.8 操作系统设计的第7条哲学原理：时空转换——沧海桑田之变

在中国的抗日战争初期，国民党屡屡战败，将大片的中国领土拱手（或在激烈战斗后放弃）让给了日本人。蒋介石说这就是以空间换时间，以等待国际形势发生对中国有利的变化。虽然，不同的人对这个观点有不同的看法，但有一点是公认的：即时间和空间可以进行转换。要么获得空间，放弃时间；要么获得时间，放弃空间。

这种时空转换在操作系统里面比比皆是。最显著的例子就是页表的实现。由于页表的尺寸通常太大，占用内存过多，我们便将页表分级，只保留一部分页表在内存，而其他部分放置于磁盘上。这样页表所占空间大为减少。但付出的代价就是时间成本：从虚拟地址转换为物理地址需要经过多级转换，从而导致转换时间增加。而为了控制这种时间的增加，操作系统又使用了快表来提升转换的速度。而快表付出的代价不是别的，正是空间的增加。

时空转换原理在算法设计中也常有体现：在散列算法的实现里，为了提高搜索的效率，我们不惜增加空间成本来获得常数时间的查找算法。

22.9 操作系统设计的第8条哲学原理：策机分离与权利分离

众所周知，在运动场上，裁判和运动员不能是同一个人，甚至不能是同一国家的人。为的

是使比赛公平。在一个国家机器里，立法机构和执法机构也不能是同一个团体，否则就会容易出现滥用法律的情况。而立法就是策略，执法就是实现机制。立法和执法分离就是策机分离哲学。美国的三权分立就是人类社会的策机分离上的典型代表。

而这种策机分离的原则在操作系统设计中得到了充分体现。不过，操作系统里面的策机分离主要不是为了公平，而是为了实现的灵活性。比如策略可以由用户指定，操作系统则是执行机制。所以有了所谓的调度算法参数化，算法在内核里，参数可以由用户指定。

例如，操作系统提供了优先级调度，但是进程的优先级却可以由用户指定。操作系统提供的是优先级调度的实现机制，而用户则负责制定优先级策略，即每个进程的优先级到底是多少，从而确保所有进程按照用户的希望进行调度。而由于策机分离，操作系统在实现优先级调度时可以随时变更实现的方法，而不会影响用户程序的执行。

跳出操作系统范畴，我们看到，策机分离在计算机领域里面到处都在体现。在程序设计领域，人们对界面的设计和对界面的实现是分开的；在计算机安全领域，对安全标准的设计和安全的实现是分开的；在通信领域，对接口规范的设计和接口的实现也是分开的。这种策机分离既保证了所有产品的一致性，又留有足够的灵活性。

22.10 操作系统设计的第9条哲学原理：简单为美——求于至简、归于永恒

爱因斯坦说过“一切都应该尽可能简单”（Everything should be as simple as possible）。在数学领域有个不成文的共识：如果一个问题有多个数学表示，那么最简单的表示通常是正确的。在人类社会里面，也是越简单的架构效率越高。而且，如果简单，其正确性也比较容易确认。而复杂的东西，要证明其正确则通常颇费周折。

就是对于个体的人来说，也应追求简单。复杂的生活让人感到疲惫，甚至会压垮人。将简单为美的哲学应用到操作系统里面，就是操作系统的设计应该越简单越好。而这种简单的哲学在操作系统里面得到了充分体现。

例如，在文件的存储方式上，我们在网型组织、树型组织、记录流、数据块流和字节流的各种选择当中，现代操作系统选择的都是最简单的字节流，如图22-8所示。

22.11 操作系统设计的第10条哲学原理：适可而止

第10条哲学原理是用来修正前面的9条原理的。即在前面所论述的9条原理的贯彻过程中，要保持一个度，适可而止。而不是无限推进，从而达到事务的反面。中国5000年的历史总结出了中庸之道，即凡事不可走极端。走极端带来的结果就是泰极生否。

例如，在简单为美的哲学下，我们不能过于简单。即使是十分注重简单的爱因斯坦，他在其著名的“Everything should be as simple as possible”的论断后面还留了一个尾巴，即“but not simpler.”很多人看到了爱因斯坦这句话的前面部分，却没有或不愿意看到其后面的这个尾巴。而这个尾巴明明白白地告诉我们：“一切都应该尽可能简单，但没有更简单”。这就给追求简

单界定了限制。而这个限制的标杆就是爱因斯坦本身。如果一个事情比爱因斯坦的东西还简单，那就是过于简单了。（从这点上可以看出爱因斯坦那种不露声色的傲慢自大：温文尔雅，却充满了骄傲。）

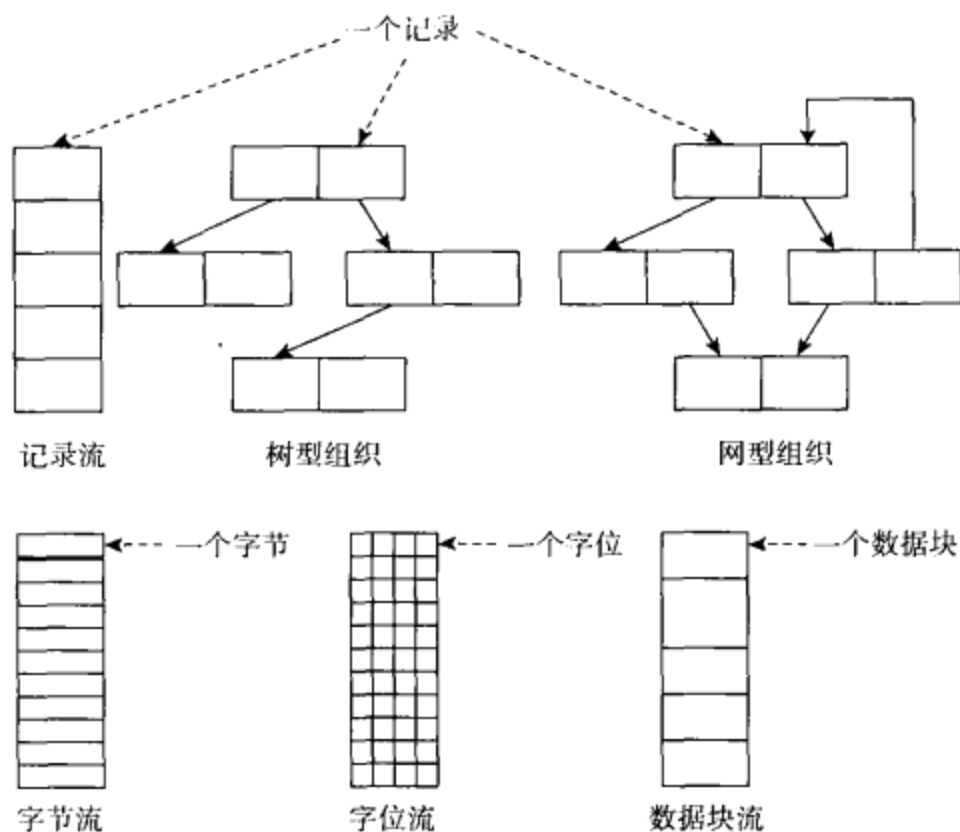


图 22-8 文件的组织形式

例如，在设计文件的组织形式时，如果使用字位流就是过于简单了，在进行页面更换时，使用随机页面替换算法就是过于简单了。

思考题

1. 你认为在效率和公平的天平上，操作系统应该将哪个放在第一位？
2. 从操作系统设计中找出其他时空转换的例子。
3. 从操作系统的其中一个核心部分里面找出层次结构的例子。
4. 你能找出更多的“留有余地”的例子吗？
5. 从操作系统技术里面找出另一个简单为美的例子。
6. 适可而止是普通人难以达到的境界，试论述如何在操作系统设计中实现此种哲学。
7. 操作系统的这些设计哲学并不仅仅只适应于操作系统一门学科，很多原则也适应于计算机的其他学科，甚至非计算机学科，你能否举出几个例子加以说明？
8. 你觉得本章列出的各种设计哲学对你理解操作系统有何帮助？
9. 你能举出操作系统里面的其他哲学原理吗？请列出你能想到的原则，并将其与人类生活联系起来。
10. 在操作系统的各种机制里面，你还能找出哪些过于简单的设计？
11. 谈谈你对操作系统技术和原理的总体感想，你觉得这些哲学设计原理在未来会改变吗？

结 语 EPILOGUE

美国密歇根大学的一位心理学教授曾经做过一个新奇的实验。他将一些蜜蜂和苍蝇放进一个敞口的大玻璃瓶里面，然后将玻璃瓶横放在桌上，底部朝向阳光。这时有趣的现象出现了：蜜蜂由于有光线感，都向着阳光的一面飞，撞到瓶底掉下来后接着还是往这个方向飞，就这样一而再，再而三，经过无数次尝试后终于力竭而亡，没有一只蜜蜂飞出玻璃瓶。而苍蝇因为没有什么光线感，胡乱飞串，所谓的无头苍蝇，一大部分都从瓶口飞了出来…

当一个人向着一个方向或者理想执著前行时，他的行为也许就是瓶里面蜜蜂的行为。我们当然应该执著，问题是我们如何能肯定我们定下的理想或追求是真实，而不是虚幻呢？我们如何断定碰到挫折是因奋力不够，还是前面有一面看不见的透明玻璃墙呢？

其实人们缺乏的往往不是执着，而是由于对方向和理想的不确定而无法执着。对理想和追求的把握需要的不仅是知识，更重要的是智慧。如何才能获得智慧呢？我想起了箴言：

“敬畏神是智慧的开端…”（箴言9章10节）

本书到此就结束了。感谢读者的执着，似乎看不完的一本书，也终于读完了。

但看完了本书，并不等于操作系统的学习就结束了。实际上，要真正掌握操作系统，对本书知识的掌握只是第一步，或者说本书提供的是学习操作系统的起点，而不是终点。

首先，本书只是从哲学原理上对操作系统予以阐述。而未过多涉及具体操作系统的实现细节。而要完全透彻地理解操作系统，这些具体实现细节则不可或缺。事实上，只有亲手设计过商业操作系统，或者亲手阅读分析过商业操作系统源代码的人，对操作系统的掌握才可能真正到位。正因为如此，本书在这里建议读者：

- 设计实现一个全功能的操作系统。
- 参加 Windows、Linux 或 Solaris 课程的学习。
- 阅读分析一个实际的商用操作系统源代码。

如果读者能够在理解本书内容的基础上完成上述三点建议中的任何一点，则对操作系统的理解将大大加深。如果能够完成上述任何两点建议，你就是一个操作系统专家了。如果能完成上述三点，则你就是操作系统领域的泰斗！

其次，本书覆盖的内容与有些读者所期待的操作系统内容并不完全一致。无容置疑，一本书究竟应该包括哪些内容是见仁见智的。而操作系统尤其如此。操作系统从无到有，在经历了许多年的发展后形成了许多子学科，而这些子学科又进一步发展成为单独的学科而从操作系统

学科中分离出来。例如：计算机网络、数据库系统、计算机安全、分布式计算等。由于这些领域均已经有了自己的课程，它们自然不应再包含在操作系统的教程里。

而就是在操作系统范围内的内容也不一定需要包括在大学本科的操作系统教程里面。由于操作系统的复杂性，其内容极为丰富，将所有与操作系统关联的内容包括在一本书里将使得整本书的内容臃肿，关键点也将淹没在繁杂的无关紧要的琐细里。因此，为了使本书重点突出，作者仅选择了对操作系统的核心内容进行讲述，其它内容留给读者自己探索。只要读者真正理解了本书的内容，在操作系统上进行进一步探索就有了坚实的基础。

当然了，由于操作系统的复杂性，细节繁多，设计出好的操作系统十分不易，甚至是不可能的。现在市面上的商用操作系统均存在这样那样的缺点，有些缺陷甚至让人难以忍受。从互联网用户对各种商用操作系统的反映来看，似乎没有什么操作系统令人完全满意。

此时，不由得想起一首英文讽刺歌《所有操作系统都很烂》(every OS sucks)。

所有的操作系统都浪费时间，从桌面到手提机
所有算盘以后的一切，只不过是一堆垃圾
从微软到苹果机，再到 Linux，Linux 真寒碜
每台计算机都死机，因为每种操作系统都很烂。
Every OS wastes your time, from desktop to the lap
Everything since the abacus, is just a bunch of crap
From Microsoft to Macintosh, to Linux Linu-nux
Every computer crashes, cause every OS sucks

由此可见，设计一个完美的操作系统是多么的困难，甚至是无法达到的。这就意味着我们在操作系统上的探索是永无止境的，虽然不时会有达到完美的感觉，但仔细思量却发现不过是自我幻觉而已。就像真理一样，似乎能感觉到，但似乎又没有把握。而本书所要达到的目标就是为读者探索操作系统提供些许指导，一个指引读者探索方向的路标。

还记得本书前言里面的那首小诗吗？它的结尾是这样的：

不要失望，
不要迷茫；
抬起头来，让我们数星星；
也许，我们能数得清；
也许，我们能看到真理的光芒
也许，这就是我们的希望……



本书只是学生探索操作系统征程上的一个路标

参考文献 BIBLIOGRAPHY

- [1] Richard McDougall, Jim Mauro. Solaris 内核结构[M]. Sun 中国工程研究院译. 北京: 机械工业出版社, 2007.
- [2] Richard McDougall, Jim Mauro. Solaris 性能与工具[M]. Sun 中国工程研究院译. 北京: 机械工业出版社, 2007.
- [3] Andrew Tanenbaum. 现代操作系统 (英文版·第2版) [M]. 北京: 机械工业出版社, 2005.
- [4] Andrew S Tanenbaum. Distributed Operating Systems[M]. Prentice Hall, 1994.
- [5] A Silberschatz, J Peterson, P Galvin, etc. Operating Systems Concepts[M]. 6th ed. Wiley, 2008.
- [6] George Coulouris, Jean Dollimore, Tim Kindberg, 等. 分布式系统: 概念与设计 (英文版·第4版) [M]. 北京: 机械工业出版社, 2005.
- [7] 多核系列教材编写组. 多核程序设计[M]. 北京: 清华大学出版社, 2007.
- [8] Mark E. Russinovich, David A. Solomon. Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000[M]. 4th ed. us; Microsoft Press, 2005.
- [9] Uresh Vahalia. UNIX internals[M]. Prentice Hall, 1996.
- [10] Johnson M. Hart. Win32 System Programming: A Windows® 2000 Application Developer's Guide[M]. 2nd ed. Addison-Wesley, 2000.
- [11] Arnold Robbins. Unix in a Nutshell[M]. 4th ed. O' Reilly Media, 2008.
- [12] IEEE Computer Magazine[J]. 2004 ~ 2009.

形而上者谓之道，形而下者谓之器。——《易经》



这是一个瞬息万变的时代。

分布式计算的脚步渐行渐远，网格计算的热潮逐步退却，云计算和云存储正慢慢揭开面纱……在所有的变化中，不变的是这些计算的支柱：操作系统！能否深刻理解它也许会决定云时代的“浮沉”。

本书从生活哲学的视角对操作系统的原理进行阐述，通过逻辑推理演绎操作系统核心技术的奥秘，讨论范围包括操作系统的所有基础内容：背景与历史、进程与线程、通信与同步、调度与死锁、分页与分段、磁盘与文件、输入与输出等。此外，作者以新颖的组织方式讲解了锁的实现、同步机制的发展逻辑、从分段到段页式的演变、多核环境下的同步与调度、操作系统设计的原则。

本书对操作系统原理的讨论充满趣味性：每一章都力求细致地阐明一个主题，将通俗的哲学原理和逻辑推理贯穿于每一个主题，构成全书的有机整体，并适当地引入计算机组成和编译器知识，揭示操作系统在程序运行中发挥的作用，把读者对操作系统的理解带到一个崭新的境界。

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

封面设计：范华明



上架指导：计算机/操作系统

ISBN 978-7-111-26642-6



9 787111 266426

定价：38.00元